Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-92-09

Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes

Gernod P. Laufkötter

März 1992

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Daimler Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Krupp-Atlas, Mannesmann-Kienzle, Philips, Sema Group Systems, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes

Gernod P. Laufkötter

DFKI-D-92-09

Diese Arbeit wurde finanziell unterstützt durch das Bundesministerium für Forschung und Technologie (FKZ ITW-8902 C4).

Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes

Gernod P. Laufkötter
Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI)
Projekt ARC-TEC.
P.O. Box 2080
W-6750 Kaiserslautern
Germany

e-mail:

laufkoet@dfki.uni-kl.de

laufkoet@informatik.uni-kl.de

Zusammenfassung

Die Implementation der Wissensakquisitionsmethode des ARC-TEC-Projektes als Software-System soll einerseits ihre Verifikation durch den Einsatz als rechnergestütztes Tool ermöglichen. Sie muß andererseits dem experimentellen Charakter der Anwendung durch hohe Flexibilität und Änderungs-/Erweiterungsfreudigkeit Rechnung tragen. Um beide Anforderungen zu erfüllen, werden hier verschiedenener Möglichkeiten zur Implementation der ARC-TEC-Methode unter Berücksichtigung vorhandener Hard- und Software-Resourcen untersucht. Ziel dabei ist es nicht, völlig unterschiedliche, inkompatible Implementationsmöglichkeiten zu vergleichen. Vielmehr wird ein portables Basissystem postuliert, das in aufeinander aufbauenden Varianten an Kapazität und Effizienz, aber auch an Entwicklungsaufwand zunimmt.

Inhalt

1 Einleitung

Durch die Entwicklung wissensbasierter Systeme für komplexe Aufgabendomänen (z.B. Maschinenbau) wird der Einsatz von Wissensakquisitionsmethoden erforderlich, die eine Modellierung sowohl der Domäne als auch der Expertise unterstützen sowie die Wissenserhebung aus unterschiedlichen Quellen ermöglichen. Ein Beitrag dazu ist die im ARC-TEC Projekt entwickelte *integrative* Wissensakquisitionsmethode. Die Implementation einer solchen Methode als Software-System soll einerseits ihre Verifikation durch den Einsatz als rechnergestütztes Tool ermöglichen. Sie muß andererseits dem experimentellen Charakter der Anwendung durch hohe Flexibilität und Änderungs-/Erweiterungsfreudigkeit Rechnung tragen. Um beide Anforderungen zu erfüllen, werden hier verschiedenener Möglichkeiten zur Implementation der ARC-TEC-Methode unter Berücksichtigung vorhandener Hard- und Software-Resourcen untersucht. Ziel dabei ist es nicht, völlig unterschiedliche, inkompatible Implementationsmöglichkeiten zu vergleichen. Vielmehr wird ein portables Basissystem postuliert, das in aufeinander aufbauenden Varianten an Kapazität und Effizienz, aber auch an Entwicklungsaufwand zunimmt.

Eingeleitet wird die Untersuchung in Kapitel zwei durch eine Bewertung zweier Software-Entwicklungs-Paradigmen, der "Structured Design"- und der "Object Oriented Design"-Methode. Der letztere dieser beiden Ansätze bildet die Grundlage für einen im Anhang angegebenen Entwurf eines objektorientierten Systems-Life-Cycle, der in den Kapiteln drei und vier als Anleitung zu objektorientierter Analyse und Design dient.

Zuvor erfolgt in Kapitel drei eine kurze Schilderung der Wissensakquisitionsmethode des ARC-TEC-Projektes, die aber keinen Anspruch auf Vollständigkeit erhebt; es wird auf Literatur verwiesen, die ausschließlich die Beschreibung dieser Methode zum Inhalt hat. Auf ihr basieren auch die im zweiten Abschnitt des Kapitels drei besprochenen Ergebnisse einer objektorientierten Analyse der Wissensakquisitionsmethode.

Kapitel vier versucht dann ein in einzelnen Beispielen dokumentiertes Systemdesign. Das *globale* Design behandelt zunächst grundlegende Gesichtspunkte für den Aufbau des Systems. Das darauf folgende *detailierte* Design erörtert Einzelheiten des Systemaufbaues, z.B. die Entwicklung einer Vererbungshierarchie.

Kapitel fünf löst sich schließlich von der Anleitung durch den objektorientierten Software Life Cycle. Es werden dort zunächst Prinzipien erläutert, die teilweise die Designentscheidungen aus Kapitel vier ergänzen, teilweise aber auch auf konkrete Gesichtspunkte der Codierung des Systems Bezug nehmen. Dem folgt eine Beschreibung

dreier, aufeinander aufbauender Implementationsvarianten. Im Hinblick auf die Anforderungen an ein Wissensakquisitionswerkzeug für komplexe Domänen sowie eine stärkere Nutzung der vorhandenen Resourcen bildet die Realisierung als System mit einem lokal lauffähigem Kern und verteilten Komponenten den Schwerpunkt der Diskussion dieser Varianten.

Der praktische Teil der Arbeit hat die Schaffung einer Software-Grundlage zur Realisierung von verteilten LISP-Anwendungen zum Gegenstand. Die Ergebnisse dieser Arbeit werden in Kapitel 5.4 angesprochen.

Eine Diskussion von Implementationsmöglichkeiten ist stets von Randbedingungen wie etwa Hard- und Software-Resourcen abhängig, die sich im Verlauf einer Untersuchung verändern können. Soweit solche Änderungen nach Abschluß der Untersuchungen, jedoch vor der Fertigstellung dieses Dokumentes erfolgten, wird in gesondert gekennzeichneten "Aktuellen Bemerkungen" auf sie eingegangen.

2 Methoden des Software Engineering

Die Entwicklung des Software-Engineering hat seit den siebziger Jahren im wesentlichen zwei unterschiedliche Software-Entwicklungs-Paradigmen hervorgebracht, deren Anwendung nach Meinung ihrer Vertreter die Qualitätsanforderungen an ein Software-System erfüllen helfen soll: Die unter *Structured Development* (SD) zusammenzufassenden, klassischen Methoden der *Structured Analysis* und des *Structured Design* sowie die objektorientierte Systementwicklung (OOD, *Object Oriented Development*). Beiden Ansätzen ist zunächst eines gemein: Sie stellen nicht etwa je eine verbindliche Methode zur Lösung der adressierten Probleme bereit, sondern umfassen vielmehr Klassen von Methoden, deren Ausprägungen von Autor zu Autor mehr oder weniger stark variieren. Der Vergleich einzelner Methoden dieser Ansätze ist Gegenstand zahlloser Veröffentlichungen; die Vielzahl der Ergebnisse rechtfertigt die Behauptung, daß bei feststehender Wahl des grundlegenden Paradigmas die Entscheidung für eine spezifische Vorgehensweise nicht zuletzt eine Frage des persönlichen Geschmacks darstellt. Hier wird daher lediglich eine pragmatische Entscheidung zugunsten eines der beiden Ansätze *SD* und *OOD* getroffen werden.

Eine inhaltliche Gemeinsamkeit von *SD* und *OOD* besteht für einige Veröffentlichungen in der Einordnung des Software-Entwicklungs-Prozesses in einen *Software-Life-Cycle*, in dem der Werdegang eines Systems von der Analyse des zu lösenden Problemes über die Entwicklung bis hin zur Anwendung beschrieben wird (vergl. Henderson-Sellers, Edwards 90). Läßt man die Anwendung einmal unberücksichtigt, lassen sich in diesem Life-Cycle grob die drei Phasen Analyse, Design und Implementation identifizieren, die sich mehr oder weniger stark überlappen können. Sie bilden den Rahmen für die folgende Charakterisierung von *SD* und *OOD*. Auf die *OOD* wird dabei etwas detailierter eingegangen werden, da sie die Grundlage der weiteren Arbeit bildet.

2.1 Die Strukturierte Systementwicklung

Die Grundlage der unter *SD* zusammenzufassenden Methoden ist das "Funktionale Paradigma" (Constantine 75). Zur Entwicklung eines Systems werden die von ihm zu erbringenden Funktionen in einer Bedarfs-Analyse identifiziert. Dies geschieht zunächst in der Terminologie des Benutzers. Das Ergebnis dieser Analyse ist eine *User Requirements Specification*, die in einer für den Systementwickler *und* den Benutzer

verständlichen Weise festhält, was das System leisten soll. In der anschließenden Design-Phase werden die so dokumentierten Funktionen in einer Top-Down-Strategie in Teilfunktionen immer größerer Detailiertheit aufgespalten. Diese Vorgehensweise ist als *funktionale Dekomposition* wohlbekannt (Loy 90). Henderson-Sellers und Edwards sprechen von der Interpretation des Problem-Raumes als einer Menge teilweise voneinander abhängiger Funktionen und Prozeduren und ihrer Übersetzung in den Lösungsraum. Datenstrukturen werden erst durch die auf ihnen operierenden Funktionen näher bestimmt. Ihre Modellierung mit Hilfe von Datenwörterbüchern und Datenflußdiagrammen ist daher der funktionalen Dekomposition untergeordnet. Jüngere Veröffentlichungen von *SD*-Methoden betonen allerdings auch die Gleichrangigkeit der Modellierung von Funktionen und Daten und ihrer Übersetzung in den Lösungsraum (Cutts 88, Jackson 83). Auch diese Ansätze behalten aber die strenge Trennung zwischen Prozeduren und Daten bei.

Spätestens in der Implementationsphase des traditionellen *Software-Life-Cycle* wird eine eine Übersetzung der aus der Analyse- und der Design-Phase resultierenden Funktionen und Prozeduren in eine Programmiersprache erforderlich. Die gängigen imperativen Programmiersprachen unterstützen zwar das funktionale Paradigma. Ihre Möglichkeiten, Funktionen in einer der Terminologie des Benutzers naheliegenden Weise auszudrücken, sind aber zumeist sehr beschränkt. Das fertige System als Ausgabe der Implementationsphase ist daher das Produkt einer *Interpretation* der *User Requirements Specification* durch den System-Entwickler mit Hilfe einer Design-/Programmiersprache und als solches mit der Gefahr behaftet, den Benutzer-Anforderungen nur teilweise zu entsprechen.

2.2 Objektorientierte Systementwicklung

Seit ihrem Entstehen vor etwa 15 Jahren haben die oben erläuterten Techniken der Funktionalen Dekomposition einen hohen Grad an Akzeptanz in der Software-Entwicklung erreicht und sind zum Teil als detaillierte Methoden beschrieben worden (Cutts 88). Die Idee der objektorientierten Programmierung fand zunächst nur ihren Niederschlag in der Entwicklung entsprechender Programmiersprachen (SMALLTALK etc.) und richtete sich daher im Gegensatz zu den *Structured*-Techniken nur an die Adresse der Systemimplementation und allenfalls des Systemdesign (Loy 90). Erste Ansätze zu objektorienterten Entwicklungsmethoden werden zum Teil von ihren Autoren mit Programmiersprachen in Verbindung gebracht, die - wie etwa ADA - gar nicht als objektorientiert im eigentlichen Sinne bezeichnet werden können (Booch 86, Smovec 84).

Erst in jüngerer Zeit wurden umfassende Software-Entwicklungsmethoden vorgestellt,

die einen objektorientierten Systems Life Cycle unabhängig von irgendeiner Programmiersprache postulieren. Diese gelten allerdings als der "State of the art" der System-Entwicklung.

2.2.1 Begriffsklärungen

Die meisten der hier im Zusammenhang mit der OOD verwendeten Termini werden zwanglos im Text eingeführt; da viele der Begriffe im weiteren Verlauf der Arbeit noch häufiger eine Rolle spielen, sollen die wichtigsten hier einzeln erläutert werden.

Attribut

Ein im Kontext eines gegebenen Problems wesentliches Merkmal einer Klasse.

Dienst

Eine Methode eines Objektes, die für andere Objekte sichtbar und - durch Nachrichtenversand - aufrufbar ist.

Entität

Mit einer Entität ist hier eine beliebige Einheit eines Problemraumes gemeint, wobei das schiere Vorhandensein gegenüber dem Wesen dieser Einheit im Vordergrund steht.

Klasse

Eine generische Beschreibung gleichartiger Entitäten (-> Objekte), die die wesentlichen Attribute in Bezug auf ein gegebenes Problem umfaßt.

Methode

Funktionales Attribut einer Klasse.

Objekt

In der Terminologie der objektorientierten Sprachen ist ein Objekt eine durch Daten und Funktionen modellierte Entität. Viele Autoren abstrahieren mit dem Gebrauch dieses Begriffes von einer Unterscheidung zwischen Klasse und Instanz, wenn diese im Kontext nicht bedeutungsvoll erscheint. Hier wird dieser Begriff eher als ein Synonym für "Instanz" verwandt.

Instanz

Individuum einer Klasse.

Slot

Nicht-funktionales Attribut einer Klasse.

2.2.2 Prinzipien der *OOD*

Die Idee der *OOD* besteht darin, den *Problemraum* eines Systems durch Identifikation und Beschreibung der in ihm vorkommenden, realen Entitäten zu modellieren. Dies kann als Gegensatz zur Identifikation von Funktionen bei der *SD* betrachtet werden. Eine Entität wird durch ihre Attribute und den mit ihr zu assoziierenden Funktionen beschrieben. Welche Eigenschaften und Funktionen eine Entität des Problemraumes ausmachen, hängt davon ab, was im Rahmen des zu lösenden Problemes von ihr verlangt wird. Beispielsweise wird eine Entität `Dokument´ in der Welt der Textverarbeitung mit einer Textdatei und einer Funktion `Speichern´ assoziiert sein
Eine aus solcher Integration von Daten und Funktionen hervorgehende Einheit bezeichnet man als *Objekt*. Einige der in diesem Zusammenhang gemeinhin *Methoden* genannten Funktionen eines Objektes können als Dienste für andere Objekte einer Welt sichtbar sein. Durch eine Nachricht kann ein Objekt ein anderes auffordern, einen seiner Dienste auszuführen. Das kann sowohl die Ausführung einer Prozedur als auch die Bekanntgabe einer Information, etwa den Wert eines Attributes bzw. eines *Slots*, bedeuten. Wie ein Objekt seine Dienste implementiert, bleibt dabei verborgen; ein Objekt ist für die Bereitstellung seiner Dienste sowie für die Aufrechterhaltung seines Zustandes gewissermaßen selbst verantwortlich. Diese von Rentsch so genannte Informations-"Kapselung" (Rentsch 82) ist die bisher wohl strengste Erfüllung von Parnas´ Forderung nach *Information Hiding* in Moduln (Parnas 72).

Um Gemeinsamkeiten gleichartiger Objekte eines Problemraumes nicht redundant zu verwalten, werden solche Objekte zu einer Klasse zusammengefaßt, die die Definition der Slots und Methoden für alle ihre Individuen beinhaltet. So eine Klasse kann eine Superklasse (einfache Vererbung) oder mehrere Superklassen (multiple Vererbung) besitzen, deren Eigenschaften sie - und damit alle ihr angehörenden Individuen - erbt und für den eigenen Bedarf spezialisieren und erweitern kann. Haben bspw. verschiedenartige Klassen einige Methoden und Slots gemeinsam, so können sie diese von gemeinsamen Superklassen ererben. Diese sollten allerdings sinnvolle Datenabstraktionen darstellen, nicht etwa eine zufällige Sammlung von Methoden und Slots. Das Prinzip der multiplen Vererbung trägt auf diese Weise dem Umstand Rechnung, daß auch Entitäten der realen Welt i.a. verschiedenen Kategorien gleichzeitig zugeordnet werden können.

Wie eingangs erwähnt, war der objektorientierte Ansatz zunächst ein reines Programmier-Paradigma. Durch Arbeiten zu objektorientierter Analyse und Design (Shlaer, Mellors 89; Booch 89; Edwards, King, Winblad 90) kann aber inzwischen von Methoden zur objektorientierten System-Entwicklung die Rede sein. Diese teilweise sehr detailierten

Methoden sollen hier aber nicht im Einzelnen diskutiert werden. Eine ausführliche Beschreibung des Objekt-Paradigmas findet sich in (Edwards, King, Winblad 90). Die Beschreibung eines objektorientierten Software Life Cycle findet sich in (Henderson-Sellers, Edwards 90). Auf sie gründet sich die dem weiteren Verlauf zugrunde liegende Beschreibung eines Life-Cycle-Modelles, das in Anhang A skizziert wird.

2.3 Bewertung der Methoden

Die beiden oben beschriebenen System-Entwicklungs-Paradigmen sind von verschiedenen Autoren verglichen und bewertet worden. Einige Veröffentlichungen unternehmen auch den Versuch der Integration beider Ansätze etwa durch ihren Einsatz in verschiedenen Phasen eines Software-Life-Cycle-Modelles. Die Diskussion darüber hält zur Zeit an und scheint in absehbarer Zeit keinen Konsensus herbeizuführen. Einen bewertenden Vergleich des objektorientierten Paradigmas mit dem der funktionalen Dekomposition als Entwicklungsstrategie liefert Patrick H. Loy in einer Veröffentlichung von 1990. Loy diskutiert unter anderem zwei der am häufigsten von Verfechtern der objektorientierten Entwicklung vorgebrachten Argumente für *OOD*. Das erste beinhaltet die schwer zu verifizierende Behauptung, daß die mit der *OOD* einhergehende, form- und strukturorientierte Denkweise im Gegensatz zur funktionsorientierten Sicht der *SD* die *natürlichere* sei. Das zweite Argument sieht in der Bewegung vom Problemraum in den Lösungsraum bei der *OOD* eine isomorphe Abbildung, die im Life-Cycle des *SD*-Ansatzes nicht gegeben sei.

Form vs. Funktion

Die Auseinandersetzung Form versus Funktion findet sich nach Loy in verschiedenen Variationen auch in anderen Gebieten der Forschung bis hin zur Anthropologie. Sie ist dort wesentlich älter und nach wie vor nicht entschieden. Es ist wohl auch nicht Aufgabe des Software Engineering, zu klären, ob ein "Ding an sich" ein Objekt ist oder nicht. Nach Ward und Mellor muß daher im Einzelfall entschieden werden, welcher Ansatz für ein Problem der "natürlichere" ist. Im Falle der integrativen Wissensakquisitionsmethode des ARC-TEC Projektes kann diese Entscheidung zugunsten der *OOD* gefällt werden:
Eine Implementation der Methode soll Wissensingenieuren und Experten interaktiv helfen, Wissens*einheiten* aus vorgegebenen Quellen zu identifizieren und schrittweise zu formalisieren. Das System unterstützt sie dabei durch eine geeignete *Repräsentation* der "Welt" der integrativen KA-Methode (KA für Knowledge Acquisition). Die vom System zu erbringenden Funktionen sind eng mit den Objekten dieser KA-Welt verknüpft. Eine formalisierte Wissenseinheit ermittelt im geplanten System beispielsweise selbst ihre Quelle oder eine ihr zugeordnete Kategorie eines Expertisemodelles (siehe Kapitel 3.1),

um diese dem Benutzer auf Anfrage mitzuteilen. Dieser Vorgang soll daher auch als Methode einer Wissenseinheit implementiert werden. Es ließe sich eine Reihe ähnlicher Beispiele etwa bezüglich der Repräsentation von Quelltexten (siehe Kapitel 3.1) angeben, die belegen, daß es sich bei der *OOD* hier tatsächlich um einen natürlichen Ansatz handelt.

Isomorphie

Constantine, ein "Papst" des SD-Ansatzes bestreitet laut Loy die größere Isomorphie des Prozesses der objektorientierten Entwicklung. Dies gelte allenfalls für Spielzeugbeispiele in Lehrbüchern. Tatsächlich kann aber das Objekt-Wörterbuch der Analyse- bzw. Design-Phase direkt in eine objektorientierte Programmiersprache umgeschrieben oder sogar in ihr verfasst werden, ohne unverständlich zu werden. Das gleiche gilt für Client-Server-Beziehungen, Part-of Beziehungen und die Vererbungsstrukturen. Die Ergebnisse einer Phase des objektorientierten Life-Cycle bleiben in der Repräsentation der folgenden Phase explizit, während die Information aus Datenfluß-Diagramm und Datenwörterbuch in der *SD* mit den klassischen Sprachen mehr implizit codiert werden. Die Gefahr von Fehlinterpretationen der Ergebnisse aus Analyse und Design ist daher bei der *OOD* kleiner.

Weitere, möglicherweise schwerer wiegende Argumente für einen objektorientierten Ansatz liegen vor allem in technischen Vorteilen objektorientierter Sprachen. Neben dem Vererbungsmechanismus sind es vor allem die Polymorphie von Nachrichten und die damit zusammenhängende dynamische Bindung bei Methodenaufrufen, die Systemdesign und -Implementation übersichtlicher und einfacher gestalten.

Polymorphie von Nachrichten

Methoden, die auf unterschiedlichen Datenobjekten konzeptuell ähnliche Operationen ausführen, können (und sollen!) in einer OO-Sprache gleiche Namen tragen. Ein und dieselbe Nachricht kann, an verschiedene Objekte gesandt, die Ausführung unterschiedlichster Aktionen zur Folge haben. Dies unterstützt vor allem die Informationskapselung: Der Sender einer Nachricht brauch kein detailiertes Wissen über den Empfänger zu besitzen, außer daß dieser eine entsprechende Methode besitzt. In einer herkömmlichen Sprache wie C müssen Funktionen mit unterschiedlichen Namen geschrieben oder innerhalb eines Funktionsrumpfes umständliche Fallunterscheidungen getroffen werden. So benötigen entweder der Aufrufer einer Funktion oder die Funktion selbst das Wissen über die spezielle Behandlung verschiedener, jedoch konzeptuell ähnlicher Datenobjekte.

Dynamische Bindung vs. statische Bindung

In herkömmlichen, gewöhnlich im Zusammmenhang mit SD-Methoden zum Einsatz kommenden Programmiersprachen erfolgt die Bindung eines Funktionsaufrufes an eine auszuführende Funktion *vor* Ausführung eines Programmes und kann zur Laufzeit nicht revidiert werden. Im Gegensatz dazu wird beim Versand einer Nachricht in einer OO-Sprache die zur Ausführung kommende Methode in Abhängigkeit des empfangenden Objektes ausgewählt. Diese dynamische Bindung erlaubt eine gegenüber herkömmlichen Sprachen wesentlich flexiblere Programmierung.

3 Objektorientierte Analyse für die integrative Wissensakquisitionsmethode des ARC-TEC-Projektes

Im folgenden wird die erste Phase des objektorientierten Systems-Life-Cycle (siehe Anhang A) auf die KA-Methode des ARC-TEC-Projektes angewandt. Zur Motivation erfolgt zunächst eine kurze Beschreibung der Methode; die Analyse gründet sich auf die ausführliche Dokumentation des Verfahrens in (Schmalhofer, Kühn et al. 90), (Tschaitschan 91) und (Schmalhofer, Schmidt 91), (Schmidt 91).

3.1 Die integrative Wissensakquisitionsmethode

Bei der Wissensakquisition zur Entwicklung komplexer, wissensbasierter Systeme ist die Nutzung unterschiedlicher Wissensquellen unerläßlich, vergl. (Schmalhofer, Kühn et al. 90). Eine Wissensakquisitionsmethode (im folgenden: KM, Knowledge Acquisition Method), die dieses Erfordernis unterstützt, muß einer Reihe von Anforderungen genügen:
Das zu erhebende Wissens ist in verschiedenen Quellen auf unterschiedlichste Art repräsentiert; die KM muß diesen Repräsentationen durch geeignete Erhebungsverfahren gerecht werden. Desweiteren kann das erhobene Wissen, insbesondere wegen seiner Herkunft aus verschiedenen Quellen, Widersprüche und Redundanzen bezüglich der adressierten Anwendungsprobleme aufweisen. Diese müssen auf angemessene Art eliminiert werden. Ein weiterer Punkt betrifft die Sicherung der Relevanz und der Suffizienz des erhobenen Wissens: Wissenseinheiten, die zur Lösung der Zielprobleme nicht beitragen können, sollten bereits in einer frühen Phase des Akquisitionsprozesses erkannt werden. Es ist ebenso notwendig, Wissenslücken rechtzeitig aufzuspüren und zu schließen.

In der integrativen KM des ARC-TEC-Projektes wurden zwei Systeme konzipiert, die - miteinander kombiniert - einen Beitrag zur Lösung der angesprochenen Probleme leisten sollen: Die Case Oriented Knowledge Acquisition Method (COKAM+) und das Case Expierience Combination System (CECoS). Die Verfahren sollen die integrierte Erhebung von Wissens aus drei verschieden Wissensquellen ermöglichen. Diese Quellen sind namentlich Texte, Archive früher gelöster Fälle und nicht zuletzt das Wissen und die Erfahrung von mitarbeitenden Experten.
COKAM+ erhebt unter Federführung eines Experten und eines Knowledge Engineers

Wissenseinheiten aus Lehrbuchtexten, die zur Erklärung zuvor ausgewählter Fälle der Domäne eingesetzt und zu diesem Zweck durch Wissen des Experten ergänzt und schrittweise formalisiert werden.
In CECoS wird mittels eines vollständigen Paarvergleichs der bereits in COKAM verwendeten Fälle eine Hierarchie von Produktionsklassen erstellt. Die einzelnen Klassen dieser Hierarchie können dann anhand vom Experten anzugebender Merkmalsbeschreibungen unterschieden werden. Die Hierarchie dient zusammen mit den Merkmalen zur Ermittlung von Problemklassen, die die Assoziation neuer, verwandter Probleme mit schon gelösten erlauben sollen.
Beide Verfahren bestehen grundsätzlich aus einer Auswahlphase zur Identifikation geeigneten Materials und einer Anwendungsphase zur Kombination des Wissens aus den verschiedenen Quellen.

3.1.1 COKAM+

3.1.1.1 Die Auswahlphase in COKAM+

Ein Experte der Anwendungsdomäne wird zunächst mit der Auswahl von Textmaterial - z.B. Lehrbücher oder wissenschaftliche Papiere - betraut. Das Textmaterial soll Wissen enthalten, das für die adressierten Probleme von Relevanz ist. Unabhängig davon bemüht sich ein Knowledge Engineer um die Auswahl einer Reihe geeigneter Anwendungsfälle. Unter einem Fall versteht man dabei eine Problemstellung zusammen mit ihrer - nach Möglichkeit mustergültigen - Lösung.

In einem weiteren Schritt der Auswahlphase markiert der Experte in den gewählten Texten einzelne Textstellen mit einer Länge von einem bis wenigen Sätzen, in denen nach seiner Meinung das für die Aufgabenstellung relevante Wissen repräsentiert ist. Die markierten Textstellen bilden die sogenannten Wissenseinheiten, die in einer informalen Wissensbasis gesammelt werden. In der rechnergestützen Durchführung der Methode soll der Experte bei dieser Aufgabe durch ein Latent-Semantic-Indexing-Verfahren unterstützt werden (Schmidt 91; Dumais et al. 88). Dieses Verfahren liefert auf eine aus Schlüsselworten formulierte Anfrage hin Textpassagen, die zur Anfrage in einem inhaltlich sinnvollen Zusammenhang stehen. Eine kurze Beschreibung des LSI-Verfahrens findet sich in Abschnitt 5.4.3.2.

3.1.1.2 Dekontextualisierung und Formalisierung von Wissenseinheiten

Zur späteren Verwendung der Wissenseinheiten in einem Inferenzsystem müssen zwei ihrer wesentlichen Mängel behoben werden: Zum einen enthalten die aus einem Ausgangstext übernommenen Einheiten häufig Kontextbezüge, die ihre Aussage

unverständlich werden lassen. Diese müssen in einem Dekontextualisierungsprozeß eliminiert werden. Zum anderen müssen die Wisseneinheiten aus ihrer natürlichsprachlichen Repräsentation in eine formale Darstellung überführt werden. Für beide Maßnahmen wird noch an geeigneten Automatisierungsmöglichkeiten gearbeitet.

Die Implementation der Methode sollte hier einen Mechanismus zur Verfügung stellen, der die schrittweise Durchführung der beiden Maßnahmen erlaubt. Die Quellpassage einer dekontextualisierten und formalisierten Wissenseinheit im Ausgangstext muß außerdem leicht wiederauffinbar sein, z.B. durch die Verwaltung hypertextartiger Link-Strukturen.

3.1.1.3 Assimilation von Wissenseinheiten in die Modelle der Domäne und der Expertise

Für die spätere Anwendungsphase ist es sinnvoll, die aus den vorherigen Schritten resultierende Menge von Wissenseinheiten weiter zu strukturieren: Hierarchisch gegliederte Teilaufgaben, die zur Lösung der in der Zieldomäne auftretenden Probleme zu bewältigen sind, bilden ein Modell der Domäne. Auf ähnliche Weise modelliert man das Verhalten des Experten bei der Problemlösung. Der Experte weist nun sämtlichen Wissenseinheiten eine adequate Kategorie beider Modelle zu. In der Anwendungsphase kann man mit Hilfe dieser Zuordnung herausfinden, welche Wissenseinheiten für eine zu erklärende Teilaufgabe von Belang sind.

3.1.1.4 Die Anwendungsphase in COKAM+

Zur Sicherung der Relevanz und der Suffizienz der erhobenen Wissenseinheiten ist schließlich eine Anwendungsphase durchzuführen: Der Experte erklärt die Lösungen der zu Beginn ausgewählten Fälle unter Verwendung der Wissenseinheiten, in dem er zu jedem Fall eine hierarchische Erklärungsstruktur erstellt. Wissenseinheiten, die sich in diesem Prozeß als überflüssig herausstellen, können aus der Wissensbasis gelöscht werden. Fehlen hingegen zur Erklärung bestimmter Schritte geeignete Einheiten, so müssen sie durch "Common-Sense"-Wissen des Experten ergänzt werden. Zusätzlich kann auch eine Wiederholung der vorherigen Schritte zur Ermittlung weiterer Wissenseinheiten erforderlich sein.

3.1.2 CECoS

3.1.2.1 Die Auswahlphase in CECoS

Auch für CECoS ist die Ermittlung geeigneter Fälle Voraussetzung zur Durchführung der Methode. Zweckmäßigerweise verwendet man die gleichen Fälle wie zuvor in COKAM+. Eine separate Auswahlphase für CECoS erübrigt sich dann.

3.1.2.2 Der vollständige Paarvergleich

Die gegebenen Fälle werden zufallsgesteuert zu Paaren kombiniert, die dem Experten nacheinander präsentiert werden. Für jedes der Paare gibt der Experte eine Einschätzung der Ähnlichkeit der beiden Fälle bezüglich der für sie durchgeführten Arbeitsplanung. Die Ähnlichkeit zweier Fälle wird durch natürlichzahlige Werte in einem vorgegebenen Bereich erfaßt. Aus diesen Werten berechnet eine hierarchische Cluster-Analyse eine sogenannte Produktionsklassenhierarchie, die in einem binären Baum dargestellt werden kann. Die Terminalknoten dieses Baumes repräsentieren die vorgegebenen Fälle, wobei zwei benachbarte Fälle besonders ähnlich nach der Einschätzung des Experten sind. Übergeordnete, nicht-terminalen Knoten fassen die untergeordneten terminalen und nicht -terminalen Knoten zu Produktionsklassen zusammen. Die Fälle bilden somit protoypische Mitglieder dieser Produktionsklassen und liefern als solche deren extensionale Definition. Der Zweck dieser Produktionsklassen liegt in der Möglichkeit, noch ungelöste Fälle aufgrund einer Merkmalsbeschreibung einer solchen Klasse zuordnen zu können. Ein für die Klasse vorhandener Skelettplan kann dann - nach geeigneter Verfeinerung - zur Lösung des neuen Falles eingesetzt werden.

3.1.2.3 Erklärung der Produktionsklassenhierarchie

Um eine Einordnung noch ungelöster Fälle in eine Produktionsklasse zu ermöglichen, versucht der Experte Merkmale zu finden, die die einzelnen Klassen eindeutig voneinander unterscheiden, die aber auf alle Mitglieder einer Klasse gemeinsam zutreffen. Auf diese Weise erfolgt die intensionale Definition der Produktionsklassen.

3.1.2.4 Assimilation der Merkmale in das Modell der Expertise

Der letzte Schritt in CECoS besteht darin, die erhobenen Merkmale durch eine Einordnung in das bereits in COKAM+ erwähnte Modell der Expertise weiter zu charakterisieren.

Die Beschreibung der Wissensakquisitionsmethode beschränkt sich hier auf die Aspekte, die zur Implementation eines rechnergestützten Tools von besonderem Intersse sind. Nähere Einzelheiten der Methode, insbesondere die Ansätze zur Integration des erhobenen Wissens aus COKAM+ und CECoS könne aus den eingangs genannten Quellen entnommen werde.

3.2 Objektorientierte Analyse

Die folgenden Analyse versucht, die unter 3.1 skizzierte KA (Knowledge Acquisition)-Welt durch eine Beschreibung der in ihr enthaltenen Objekte zu modellieren. Das Augenmerk richtet sich dabei zunächst auf solche Objekte der KA-Welt, deren Existenz nach obiger Beschreibung unmittelbar klar ist. Die Einführung weiterer Objekte kann später aus software-technischer Sicht sinnvoll sein.
Welche Objekte lassen sich nun in der Welt der integrativen KA-Methode identifizieren? Zunächst sind dies die Anwender des Systems, ein oder mehrere **Knowledge-Engineers** und **Experten**. Sie sollen lediglich als Benutzer zum Zweck eventuellen Mehrbenutzerbetriebes in Versuchen zu paralleler Wissensakquisition verwaltet werden. Desweiteren gibt es **Quelltexte** in unserer KA-Welt. Aus ihnen werden die **Knowledge-Units** erzeugt, die zusammen mit **Fällen**, **Erklärungsstrukturen**, **Domänen-Modell** und einem **Modell der Expertise** eine höhere Struktur bilden, die dem Benutzer als **informale Wissensbasis** bewußt ist. Weitere Bestandteile der informalen Wissensbasis sind **Produktionsklassen**, die als Output der Anwendung von CECoS in einer **Produktionsklassen-Hierarchie** enthalten sind. Zur Charakterisierung der Produktionsklassen werden in CECoS außerdem **Features** erhoben.

In einer prototypischen Implementierung der KA-Methode hat sich die Einordnung der in Bearbeitung befindlichen Teilmenge der Knowledge-Units aus der Wissensbasis in einen Bildschirm-Karten-Stapel als sinnvoll erwiesen. Ihm fällt die Aufgabe zu, größere Mengen von Units am Bildschirm möglichst übersichtlich darzustellen. Als "elektronischer Karteikasten" ist dieser **Unit-Stack** fester Bestandteil der KA-Welt.

Exemplarisch werden hier die Objekte **Quelltext** und **Knowledge-Unit** auf ihre Eigenschaften hin untersucht und als Klassen beschrieben[1]. Sie bilden zentrale Bestandteile der KA-Methode und eignen sich daher in besonderem Maße als Beispiel. Das weitgehend vollständige Objekt-Wörterbuch als Ergebnis der Analyse findet sich in Anhang B

Abbildung 3.2.1 zeigt die Darstellung eines Objektes *Source-Text* mit den Slots und Diensten, deren Existenz bereits in der Analyse evident ist. In Tabelle 3.2.1 sind die einzelnen Slots und Dienste mit ihrer Bedeutung aufgelistet.

Analog zu einem Quelltext wird eine *Knowledge-Unit* beschrieben. Abbildung 3.2.2 zeigt das Objekt im Zusammenhang.

---

[1]Ein Objekt ist hier nicht identisch mit seiner Klasse; jedoch kann die *exemplarische Beschreibung* eines Objektes als Klassendefinition aufgefaßt werden.

Klasse *Source-Text*

Slot	Beschreibung
Textfile	Die mit einem Quelltext assoziierte Eingabe-Textdatei
Title	Der Titel eines Quelltextes
Dienst	
hide	Die zu *show* inverse Funktion. Ein Quelltext-Objekt verschwindet mit all seinen dargestellten Komponenten vom Schirm. Die interaktiven Dienste des Objektes stehen dem Benutzer bis zu einem erneuten *show* nicht zur Verfügung
marked-text	Ein Aufruf dieses Dienstes soll im Quelltext markierten Text als Zeichenkette an das aufrufende Objekt, etwa eine Unit, liefern und die Markierung rückgängig machen.
mark-text	Dem Anwender wird über diesen Dienst das Markieren von Textpassagen im angezeigten Text ermöglicht
show	Ein Quelltext-Objekt, das diese Nachricht erhält, soll sein Textfile zur Anzeige bringen und dem Benutzer seine interaktiven Dienste zur Verfügung stellen
show-text-at-position	Veranlaßt das angesprochene Objekt, den Text an einer als Argument spezifizierten Position anzuzeigen
structure	Es soll eine wie auch immer geartete Textstruktur - automatisch oder interaktiv - erzeugt werden und so ein komfortables Navigieren im Text ermöglichen. Die Art dieses Dienstes wird bewußt nicht näher spezifiziert, da sich in Quelltext-Objekten einmal verschiedene Text-Browsing-Tools verbergen sollen

Tabelle 3.2.1 Die Slots und Methoden der Klasse *Source-Text*

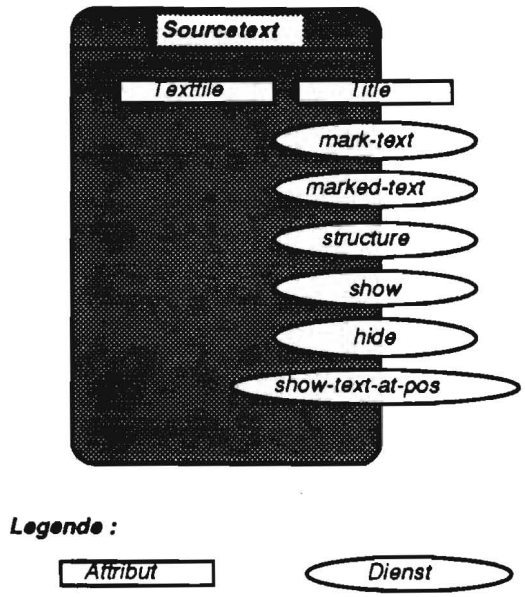

Abbildung 3.2.1: Graphische Darstellung eines Objektes *Source-Text* mit Slots und Diensten. Ellipsen, die aus dem grauen Rahmen der Graphik hervortreten, markieren für andere Objekte sichtbare Dienste.

Klasse *Knowledge-Unit*

Slots	Beschreibung
Children	Die Nachfolger einer Unit
Domain Category	Eine der Unit zugewiesene Kategorie des Domänen-Modells
Expertise Category	Eine Kategorie des Modells der Expertise, die der Unit zugewiesen wurde
Source	Quelle des Textes einer Unit. Das ist entweder ein Quelltext oder eine Unit
Text	Der aus einem Quelltext oder einer Vater-Unit kopierte Text
Title	Der Titel einer Unit
Used-for-explanation	Für eine Unit, die bereits in Erklärungsstrukturen verwendet wurde, gelten im Falle einer Änderung oder des Löschens Einschränkungen. Daher muß die Verwendung in einer Erklärung protokolliert werden.

Tabelle 3.2.2 Die Slots der Klasse *Knowledge-Unit*

Klasse *Knowledge-Unit*

Dienste	Beschreibung
add-text	Aus einem Quelltext übernommener Text soll den bisherigen Text der Unit ergänzen. Die Quelle wird in der Unit vermerkt.
create-child	Es soll eine neue Unit erzeugt werden, die markierten Text aus der angesprochenen Unit erbt. Die neue Unit wird den Wert des Slots *Children* ergänzen.
delete	Eine Unit, die diese Nachricht empfängt, hat dafür Sorge zutragen, daß sie und alle Referenzen auf sie aus der informalen Wissensbasis gelöscht werden. Dabei dürfen keine "Waisenkinder" entstehen, d.h. Kinder der Unit müssen die Quellenangabe ("Source") der zu löschenden Unit erben
get-domain-category, ***get-expertise-category***	Eine als Argument übergebene Domänen- bzw. Expertisen-Kategorie wird der Knowledge-Unit zugeordnet.
hide	Der zu *show* (siehe unten) inverse Dienst. Die Unit soll vom Bildschirm verschwinden.
mark-text	Der Anwender muß zum Zweck der Dekomposition der Unit in formalere Nachfolger die Möglichkeit haben, Text in der Unit zu markieren
set-title	Setzen/Ändern des Titels
show	Die Unit soll am Bildschirm in einer geeigneten Position erscheinen
show-children	Die Nachfolger einer Unit sollen bekanntgegeben werden
show-domain-category, ***show-expertise-category***	Eine Unit soll die ihr zugeordnete Domänen- bzw. Expertisen-Kategorie bekanntgeben
show-source	Die angesprochene Unit soll ihre Quelle bekanntgeben

Tabelle 3.2.2 Die Dienste der Klasse *Knowledge-Unit*

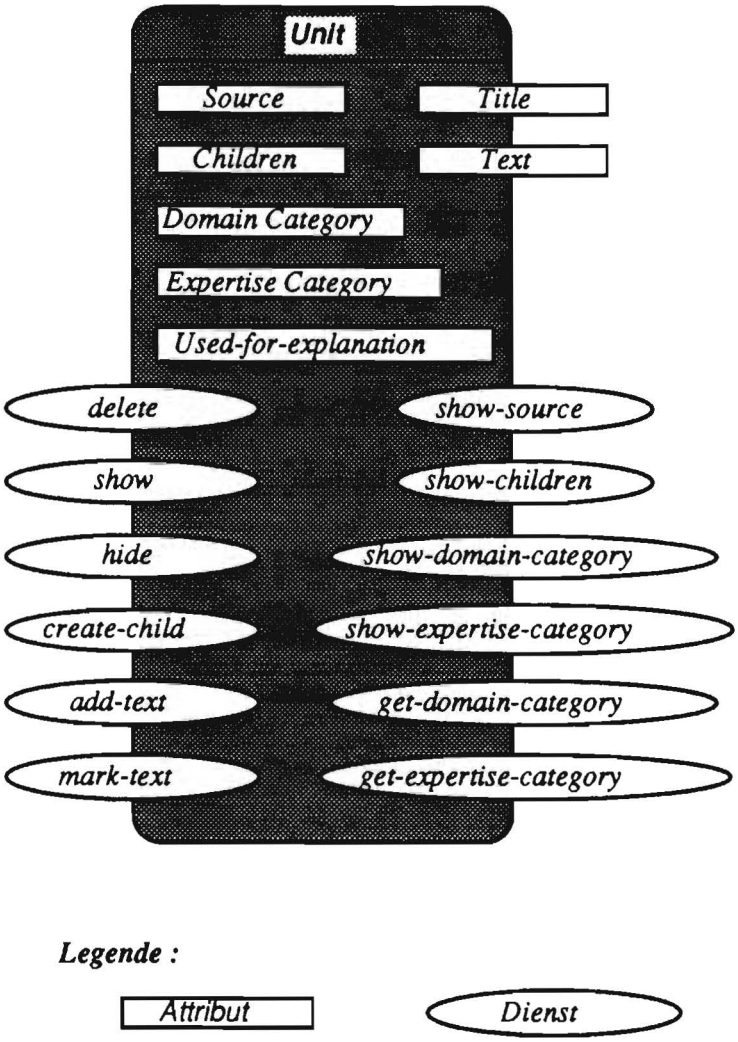

Abbildung 3.2.2: Graphische Darstellung der Klasse *Knowledge-Unit*.

Die genaue Beschreibung der restlichen, oben identifizierten Objekte findet sich in Anhang B. Auf ihrer Basis werden im folgenden Kapitel die einzelnen Schritte der Design-Phase vorgenommen.

4 System-Design

In den folgenden Abschnitten werden für einige der im Objektwörterbuch der Analyse aufgeführten Klassen die einzelnen Schritte des objektorientierten Designs (siehe Anhang A.b) durchgeführt. Da der Umfang der Dokumentation eines solchen Design mit zunehmender Detailiertheit kombinatorisch wächst, wird die Durchführung der einzelnen Designschritte auf ausgewählte Beispiele beschränkt. Auf spezielle, für die jeweiligen Implementationsalternativen erforderlichen Erweiterungen der hier vorgestellten Ergebnisse des Designs wird in den entsprechenden Abschnitten hingewiesen.

4.1 Globales Design

Entsprechend des in Anhang A vorgestellten objektorientierten *Systems Life Cycle* konzentriert sich das globale Systemdesign auf die Identifikation von Interaktionen zwischen in der Analyse identifizierten Objekten des Problemraumes. In Phase der Analyse wurden zu jedem Objekt die mit ihm assoziierten Methoden geschildert (Anhang B); im weiteren Verlauf gilt es, die durch ein gegebenes Objekt getätigten Aufrufe der Methoden anderer Objekte, also die zu versendenden Nachrichen zu identifizieren. Dies wird in 4.1.3 exemplarisch für eine Reihe von Objekten durchgeführt, die an der Erzeugung einer Produktionsklassenhierarchie im Rahmen von CECoS beteiligt sind. Der Grund für die Auswahl dieses Beispiels liegt in dem Wunsch, den in Prolog implementierten Prototyp von CECoS zum Zwecke der Integration mit COKAM+ objektorientiert zu reimplementieren. Zunächst aber werden einige Überlegungen angestellt, die für das globale Design des geplanten Systems von grundlegender Bedeutung sind.

4.1.1 Integration einer Benutzerschnittstelle

Da die Anwendung des gesamten geplanten Systems als Implementation der integrativen Wissensakquisitionsmethode ein hohes Maß an Interaktion mit dem Benutzer erfordern wird, muß im Rahmen eines globalen Designs auch die Integration einer Kommunikationsschnittstelle zum Benutzer in ein Modell der Objektinteraktion erfolgen. Diese Schnittstelle soll als grapische Benutzeroberfläche realisiert werden. Sie wird im folgenden zunächst als eine Art Black-Box betrachtet, der es ermöglicht ist, nach Aufforderung durch den Benutzer Nachrichten an die Objekte der eigentlichen Applikation zu versenden sowie Anforderungen dieser Objekte zu bearbeiten. Die Gründe

für diese Betrachtung und die durch sie implizierte, scharfe Trennung zwischen Benutzerschnittstelle und Applikation werden in Kapitel 5 näher erläutert.

4.1.2 Persistente Objekte

In der Phase der objektorientierten Analyse wurde lediglich die Existenz von Objekten festgestellt und ihre eher vordergründigen Eigenschaften beschrieben. Bei der Erzeugung einer informalen Wissensbasis als Folge der rechnergestützten Anwendung einer Wissensakquisitionsmethode muß nun auch die Lebensdauer der Objekte diskutiert werden, die erhobenes Wissen oder Wissensquellen während einer Rechnersitzung repräsentieren. Selbstverständlich wird man von den Instanzen der meisten im Objektwörterbuch in Anhang B aufgeführten Klassen erwarten, das sie die Anwendung der Wissensakquisitionsmethode am Rechner "überleben" und für weiteren Anwendung zur Verfügung stehen. Im einzelnen sind es die Objekte der Klassen *Informal Knowledge Base* und ihre Bestandteile, der Klasse *Source-Text* sowie Objekte der Klasse *Case*, die nach Beending einer Session auf ein externes Medium gespeichert und später wieder geladen werden sollen. Da sich permanent gespeicherte Objekte nicht - wie der berühmte Lügenbaron - selbst "an den Haaren aus dem Sumpf" - dem Speichermedium - ziehen können, wird ein Objekt *Persistent Object Loader* eingeführt, das die zum Laden der persistenten Objekte nötigen Methoden besitzen soll. Das Prinzip ist folgendes: Erhält ein persistentes Objekt eine Nachricht mit dem Auftrag, seinen Dienst "Speichern" auszuführen, schreibt es zunächst den Namen seiner Klasse, sodann die Werte seiner Slots, etwa Lizenzen anderer Objekte, auf den Ausgabestrom. Zum Laden eines persistenten Objektes liest der *Persistent Object Loader* einen Klassennamen vom Eingabestrom und erzeugt eine noch leere Instanz der bezeichneten Klasse. Diese wird dann aufgefordert, mit ihrer Methode "Load" ihre Slots mit den auf dem Eingabestrom folgenden Werten zu initialisieren. Aufgabe des *Persistent Object Loader* ist es dabei nur, auf Verlangen des Benutzers die Objekte zu laden, die an der Spitze von Part-of-Hierarchien (siehe 4.2.1 und Anhang A, Detailiertes Design), etwa einer informalen Wissensbasis, stehen. Diese müssen dann bei Bedarf selbst für die Bereitstellung ihrer Bestandteile Sorge tragen. Um auch den Zugriff auf die dem Benutzer zur Verfügung stehenden Fälle der Domäne durch eine übergeordnete Einheit zu organisieren, wird noch die Klasse *Case Base* eingeführt. Sie soll Methoden enthalten, die dem Benutzer die Auswahl von Fällen für die fallorientierte Wissensakquisition in COKAM+ und CECoS erlauben. Die Beschreibung der zusätzlich identifizierten Klassen findet sich in Anhang B.

4.1.3 Die Erzeugung einer Hierarchie von Produktionsklassen in CECoS als Interaktion eines Benutzers mit Objekten des Systems

Objekte der folgenden Klassen sind an der Erzeugung von Produktionsklassen und ihrer Einordnung in eine Hierarchie beteiligt (siehe auch Tschaitschian 91):

Informal Knowledge Base

Case Base

Case

Case Pair

Case Pairlist

Auf das Laden einer Fallbasis und einer informalen Wissensbasis durch Interaktion des Benutzers mit dem oben kurz beschriebenen Persistent Object Loader soll hier nicht detailiert eingegangen werden; die Präsenz der betreffenden Objekte wird im folgenden vorausgesetzt. Die geschilderten Zusammenhänge sind in Abbildung 4.1.3.1 graphisch dargestellt.

Zunächst spricht ein Benutzer über ein *Graphical User Interface* den Dienst *choose-cases* der Fallbasis (siehe 4.1.2, unten) an, der ihm die Auswahl der für die Problemstellung der Anwendungsdomäne relevanten Fälle erlaubt. Die Fallbasis weist ihrerseits die Benutzerschnittstelle an, ein graphisches Dialog-Objekt mit den Titeln der zur Verfügung Fälle anzuzeigen, der ihr nach seiner Beendigung die Titel der ausgewählten Fälle zurückliefert. Daraufhin fordert sie die informale Wissensbasis auf, diese Fälle als Argumente ihres Dienstes *get-case* in Form von Fallreferenzen (Objekte der Klasse *Case-Reference*) zu übernehmen. Ein Benutzer kann die informale Wissensbasis jetzt auffordern, die zur Erzeugung einer Produktionsklassenhierarchie notwendigen Vorbereitungen zu treffen. Die Benutzeroberfläche nimmt diese Aufforderung auf geeignete Weise (i.a. durch ein Menü) entgegen und übersetzt sie in einen Aufruf des Dienstes *generate-production-class-hierarchy* der Wissensbasis. Diese erzeugt daraufhin aus den ihr zugewiesenen Fällen ein Objekt vom Typ *Case-Pairlist*, das eine Liste sämtlicher 2-Kombinationen der referenzierten Fälle sowie Methoden zur Durchführung des nun erforderlichen *Paarvergleichs* enthält. Die Wissensbasis fordert diese Paarliste nach ihrer Erzeugung auf, ihren Dienst *perform-pair-comparison* auszuführen, der durch Ansprechen der Benutzer Schnittstelle die Darstellung eines Dialoges bewirkt. Dieser stellt dem Benutzer die Dienste *show-next-pair*, *show-previous-pair* und *terminate-pair-comparison* der Paarliste sowie den Dienst *get-similarity* des jeweils dargestellten Fallpaares zur Verfügung. Ist der Paarvergleich nach

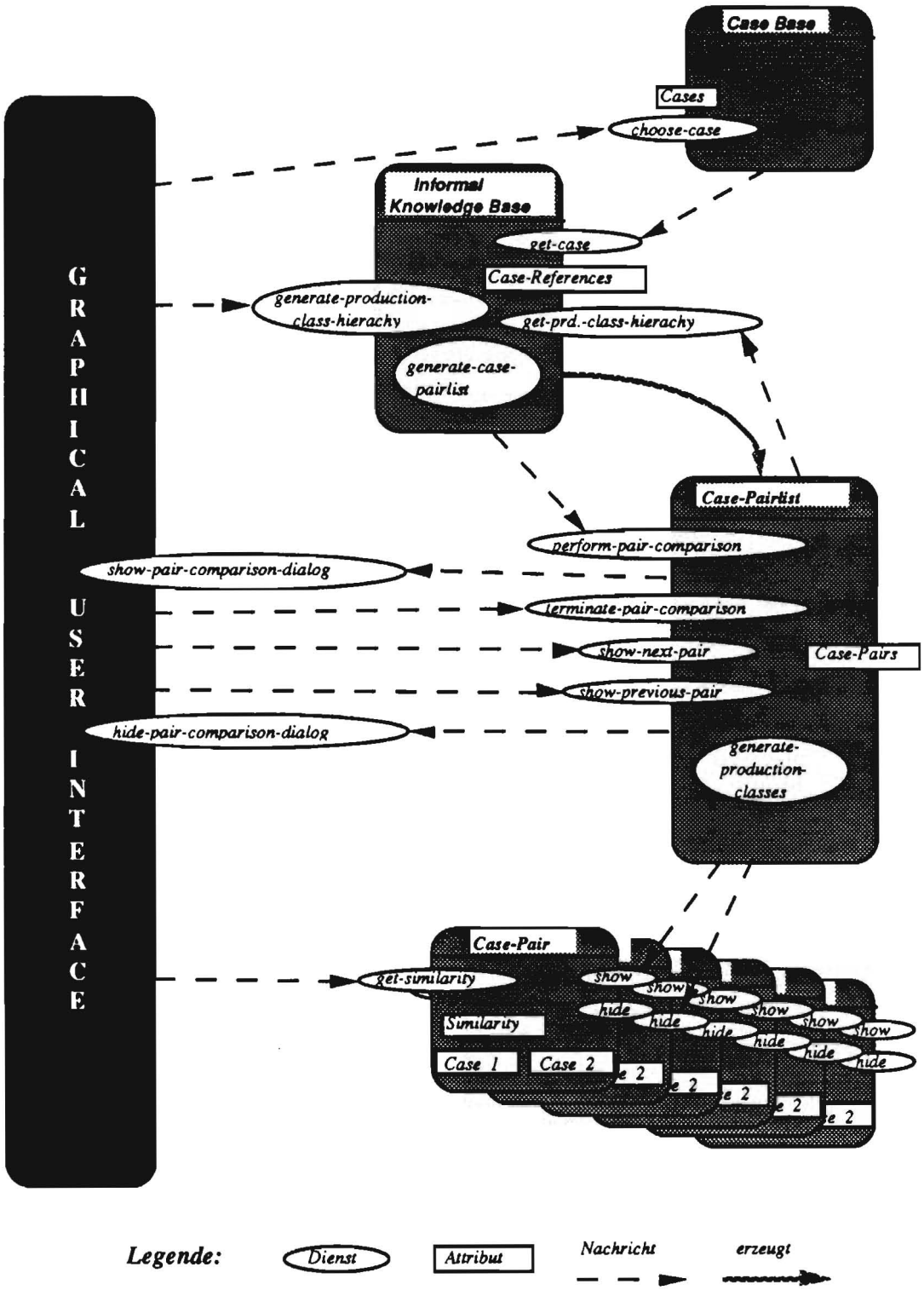

Abbildung 4.1.3.1: Vereinfachtes Interaktionsdiagramm für die an der Erzeugung einer Produktionsklassenhierarchie beteiligten Objekte.

vollständiger Durchführung beendet, erzeugt die Paarliste mit Hilfe der erhobenen Ähnlichkeitswerte die Produktionsklassenhierarchie und übergibt diese an die Wissensbasis. In Abbildung 4.1.3.1 fehlt die Darstellung der Fälle als Bestandteile der Fallpaare aus Gründen der Übersichtlichkeit.

4.1.4 Identifikation von Clustern

Die Klassen der an der Erzeugung der Produktionsklassenhierarchie beteiligten Objekte bilden ein Cluster im Sinne des Schrittes (i) der in A.c geschilderten Implementationsphase. Zur übersichtlichen Gliederung der Dokumentation des globalen Designs ist zweckdienlich, solche Cluster bereits in dieser Phase zumindest vorläufig zu identifizieren. Die Umschreibung von Clustern ergibt sich hier aus den verschiedenen Phasen der integrativen Wissensakquisitionsmethode.

Erzeugen einer Arbeitsumgebung

beteiligte, interagierende Klassen:

Case-Base
Informal-Knowledge-Base
Persistent-Object-Manager
Source-Text

Markieren von Text und Erzeugung von Knowledge-Units

Source-Text
Unit-Stack
Informal-Knowledge-Base
Unit

Erzeugen von Erklärungsbäumen

Case
Case-Base
Explanation
Informal-Knowledge-Base
Sub-Explanation
Unit
Unit-Stack

Erzeugen einer Produktionsklassenhierarchie

(siehe oben)

Angabe von Features

Case

Case-Pair

Case-Pairlist

Informal-Knowledge-Base

Feature

Production-Class

Production-Class-Hierarchy

Unit-Stack

4.2 Detailiertes Design

Aufgrund der Erfahrungen mit der prototypischen Implementation der integrativen Wissensakquisitionsmethode (Laufkötter 91; Tschaitschian 91) sind einige Ergebnisse der ersten Schritte einer detailierten Designphase bereits vorgedacht oder in den Ergebnissen der Analyse implizit enthalten. Dies gilt insbesondere für die Dekomposition von Objekten in Teilobjekte und die sich daraus ergebende Identifikation von Part-of-Hierarchien. Abschnitt 4.2.1 beschränkt sich daher auf die exemplarische Dokumentation dieser Hierarchien.

Anders verhält es sich mit der Entwicklung einer Vererbungshierarchie. Der wenig objektorientierte Aufbau des erwähnten Prototyps läßt die Übernahme von Vererbungsstrukturen nur teilweise zu. Ebenso wenig kann auf eine Bibliothek fertig implementierter, abstrakter Klassen zurückgegriffen werden (vergl. Anhang A.b). Abschnitt 4.2.2 versucht daher eine knappe, aber vollständige Synthese einer durch geeignete, abstrakte Klassen initiierten Vererbungshierarchie. Der spätere Einsatz einer Programmiersprache, die multiple Vererbung unterstützt, wird dabei vorausgesetzt.

Bei der Entwicklung der Methoden einzelner Klassen kann zu einem großen Teil auf Algorithmen der alten Implementation zurückgegriffen werden. Neue Methoden sind aus Algorithmen zur Konsistenzerhaltung in Erklärungsbäumen zu entwickeln. (Birk 91) diskutiert solche Algorithmen ausführlich. Das Prinzip der Entwicklung weiterer Methoden wird mittels der in 4.2.2 angegebenen Vererbungshierarchie anhand eines Beispiels demonstriert.

4.2.1 Dekomposition von Objekten

Die in diesem Abschnitt exemplarisch dokumentierten Part-of-Hierarchien gehen größtenteils aus den Ergebnissen der Analyse hervor. "Part-of" ist dabei wie folgt zu verstehen:

Eine Instanz A einer Klasse AK ist Teil einer anderen Instanz B einer (Klasse BK), gdw. der Wert eines Slots der Instanz A als ein Verweis auf Instanz B interpretiert werden kann.
Dies gelte unter der Voraussetzung, daß Instanzen der Klasse BK im Kontext der Anwendung als Bestandteile von Instanzen der Klasse AK sinnvoll sind. Die Klassen AK und BK müssen dabei nicht notwendig verschieden sein.

Die Interpretation eines Slotwertes als Verweis auf eine Instanz bedeutet dabei, daß der Wert sowohl ein *Pointer* in der Semantik der zugrundeliegenden Sprache auf die dereferenzierte Instanz sein kann als auch ein Name oder eine Lizenz des dereferenzierten Objektes, die es ermöglicht, das Objekt im Speicher oder auf einem externen Medium zu identifizieren. Oft ist es sinnvoll, von beiden Möglichkeiten Gebrauch zu machen. Daher sollen alle bisher identifizierten Objekte noch einen Slot *License* erhalten; auf die Gründe hierfür wird in Kapitel 5 näher eingegangen.

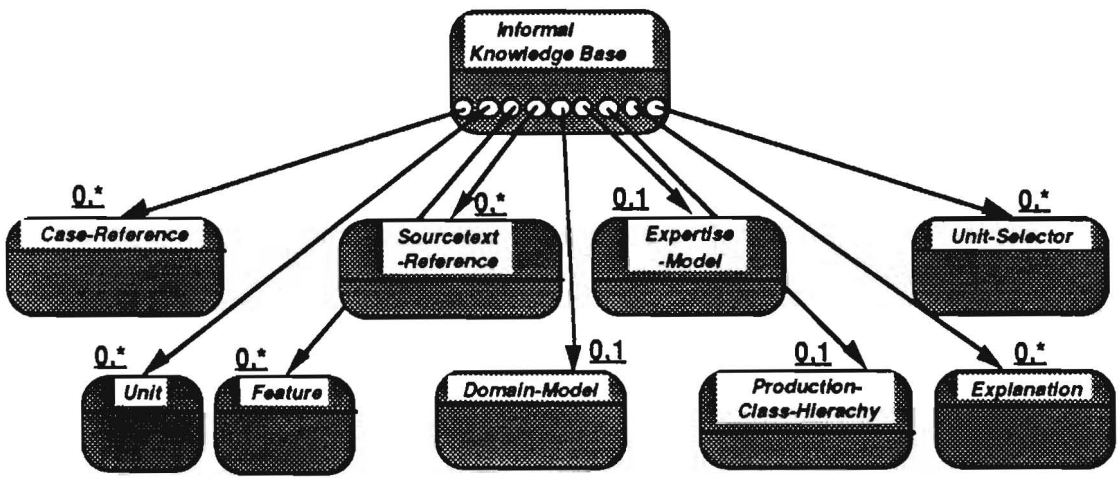

Abbildung 4.2.1.1 Der Aufbau einer informalen Wissensbasis. Zu den einzelnen Bestandteilen ist jeweils ihre mögliche Vielfachheit angegeben. Bspw. kann eine informale Wissensbasis *kein* oder *genau ein* Modell der Expertise enthalten, jedoch beliebig viele Units, Features etc.

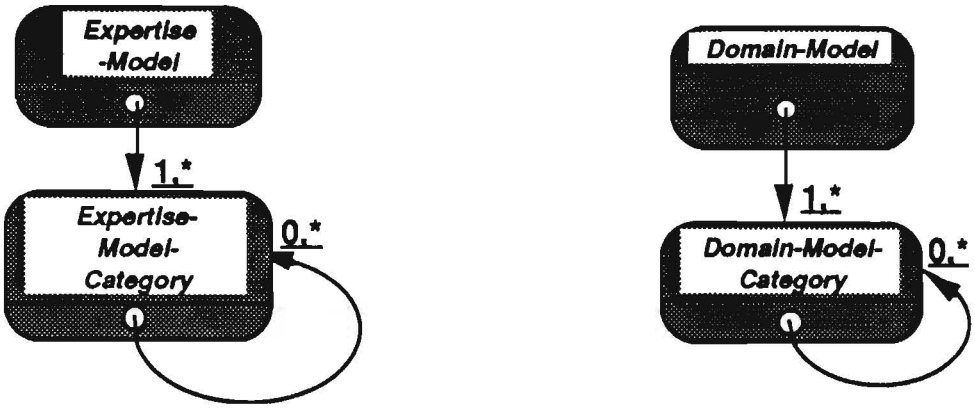

Abbildung 4.2.1.2 Der Aufbau des Expertisen- und Domänenmodelles. Die gekrümmten Pfeile deuten daraufhin, daß eine Objekt der Klasse *Domain (Expertise) Model Category* beliebig viele Objekte der gleichen Klasse enthalten kann. Eine zyklischer Aufbau soll ausgeschlossen sein; dies geht aus der Graphik nicht hervor.

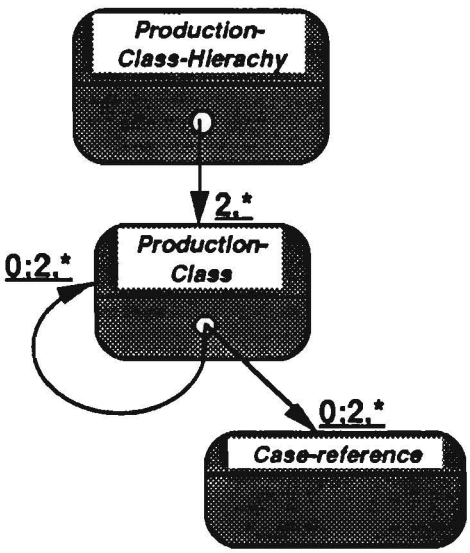

Abbildung 4.2.1.3 Diese Darstellung des Aufbaus von Produktionsklassenhierarchien ist folgendermaßen zu interpretieren: Ein Objekt *Production-Class* enthält entweder weitere Objekte dieser Art (mindestens zwei) oder als Blatt des Hierarchiebaumes mindestens zwei *Case-References*. Wie schon bei den Modellen der Abbildung 4.2.1.2 sollen Zykel ausgeschlossen sein.

Die Abbildungen 4.2.1.1 - 4.2.1.3 illustrieren den Aufbau der wesentlichen Part-of-Hierarchien. Die offensichtliche Ähnlichkeit der Strukturen des Domänenmodells mit der des Expertisenmodells oder auch einer Hierarchie von Units führt bei der Entwicklung der Vererbungshierarchie zur Identifikation gemeinsamer Superklassen der an den o.g. Strukturen beteiligten Klassen. Auf ähnliche Weise tragen analoge Eigenschaften der übrigen Klassen zur Entwicklung der Hierarchie bei.

4.2.2 Die Vererbungshierarchie

In Ermangelung einer Bibliothek vordefinierter, allgemeiner Klassen muß die Identifikation gemeinsamer Eigenschaften der Klassen unserer KA-Welt zur Entwicklung geeigneter Superklassen führen. Die in den Abbildungen 4.2.2.1 und 4.2.2.2 gezeigten Hierarchien sind auf diesem Wege entstanden. Zur Beschreibung der Hierarchien wird hier aus Gründen der Verständlichkeit der umgekehrte Weg gewählt: Es werden zunächst eine Reihe allgemeinster Klassen angegeben, deren grundlegende Eigenschaften von den meisten der Klassen, von denen später Instanzen zu erzeugen sind, geteilt werden. Aus ihnen werden dann speziellere Klassen gebildet.

Klasse *Tree-Root*

Sowohl die Modelle der Expertise und der Domäne wie auch die Dekompositionshierarchien der Knowledge-Units oder eine Hierarchie von Produktionsklassen bilden Baumstrukturen, die üblicherweise eine Wurzel besitzen. Die Eigenschaft, Wurzel eines Baumes sein zu können, wird von der Klasse Tree-Root vererbt. Sie besteht im wesentlichen darin, ein Attribut *Children* zu besitzen, das den Zugriff auf die Nachfolgeknoten des Baumes erlaubt.

Klasse *Titled-Document*

Alle Bestandteile der informalen Wissensbasis und der Fallbasis besitzen Namen oder Titel. Der Slot *Title* sowie die Methoden zum Setzen, Ändern und Lesen des Titels werden von dieser Klasse vererbt.

Klasse *View-Document*

Alle Klassen, deren Instanzen eine graphische Repräsentation zur Darstellung am Bildschirm besitzen, erben die dafür notwenigen Eigenschaften von der Klasse View-Document. Sie besitzt ein Attribut, das einen Verweis auf ein Graphik-Objekt der Benutzerschnittstelle enthält und stellt die grundlegenden Versionen der Methoden *show* und *hide* zur Verfügung.

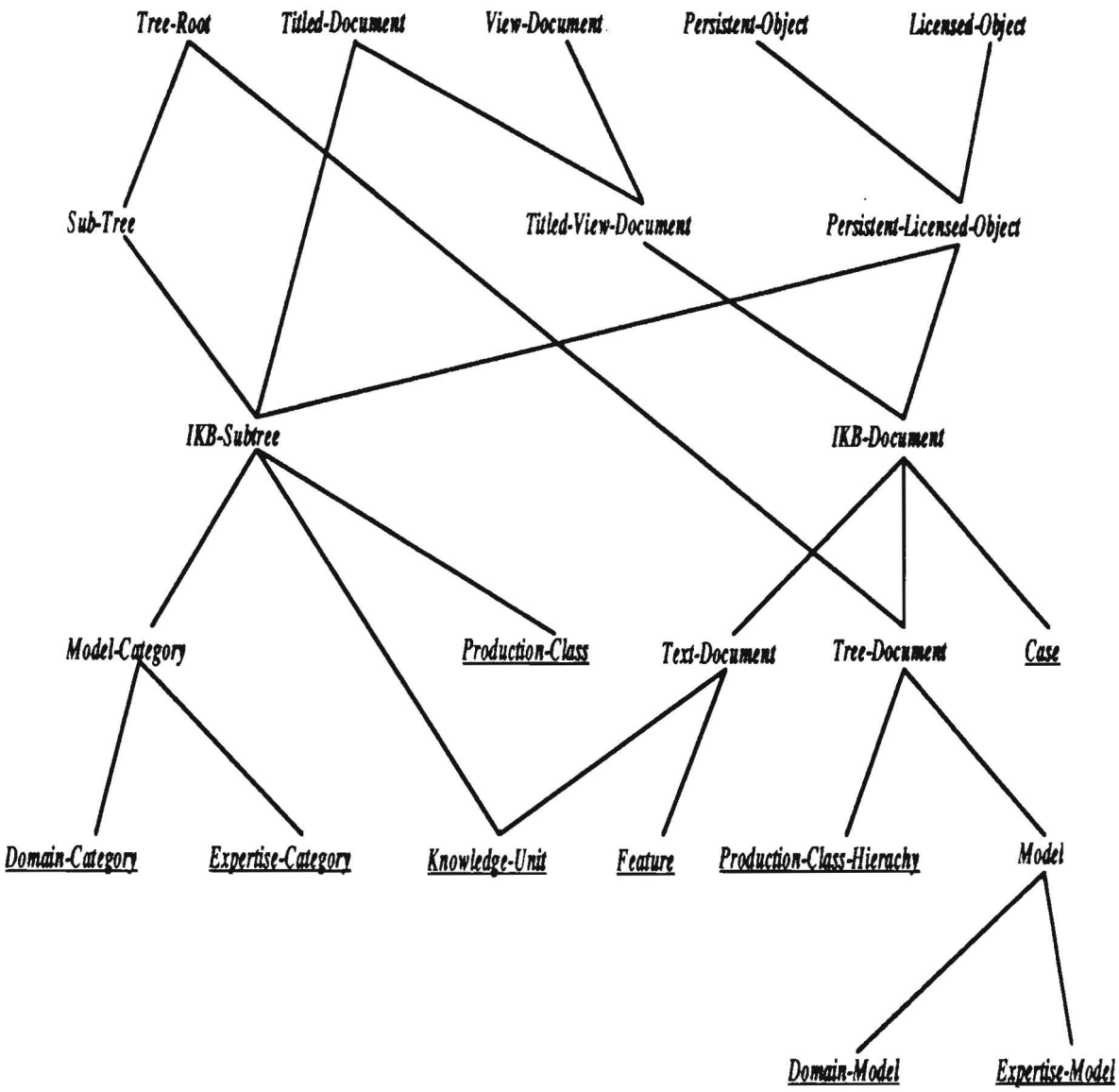

Abbildung 4.2.2.1: Eine multiple Vererbungshierarchie. Die unterstrichenen Klassennamen korrespondieren mit den in Kapitel 3. und 4. identifizierten Klassen. Nur von ihnen werden später Instanzen erzeugt.

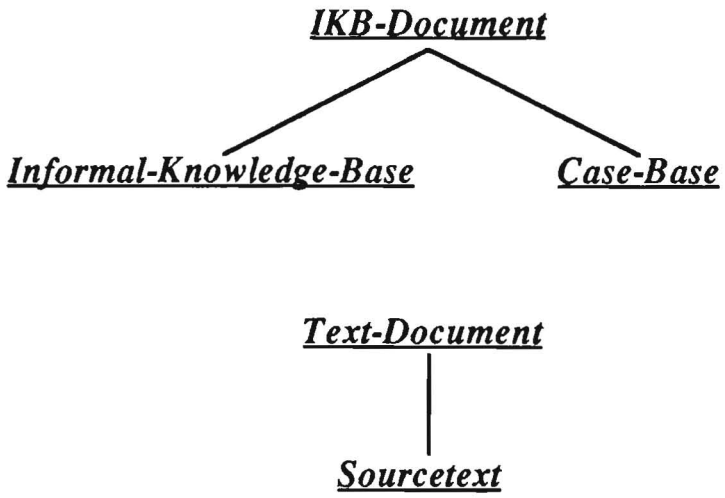

Abbildung 4.2.2.2: Instanzen der Klassen Informal-Knowledge-Base und Case-Base sind zwar keine Bestandteile einer informalen Wissensbasis, benötigen aber die gleichen, durch IKB-Document bestimmten Eigenschaften.

Klasse *Persitent-Object*

Diese Klasse definiert die Methoden *save* und *load* für alle Klassen, deren Instanzen einmal auf einem Massenspeichermedium abgelegt werden sollen. Eine Klasse, die von Persitent-Object erbt, muß ihre eigene Methode *save* besitzen, die für das Speichern ihrer privaten Slots Sorge trägt. Innerhalb dieser Methode wird dann die ererbte Version aufgerufen (siehe auch 4.2.3).

Klasse *Licensed-Object*

Die Methode zur Erzeugung von Instanzen von Subklassen der Klasse Licensed-Object wird so erweitert, daß jeder neuen Instanz eine eindeutige, unveränderliche "Lizens" in Form einer natürlichen Zahl o.ä. zugewiesen wird. Außerdem vererbt diese Klasse Methoden, die eine Liste von Instanz-Lizensen in eine Liste der assoziierten Instanzen umrechnen und umgekehrt.

Die Eigenschaften dieser allgemeinen Klassen werden nun in spezielleren kombiniert. Dies ist sinnvoll, um für eine Klasse die Liste der Superklassen überschaubar zu halten, ohne auf die Möglichkeit zu verzichten, elementare Eigenschaften allgemeinster Klassen

einzeln zu vererben. Hinter den Klassennamen sind jetzt in Klammern die Superklassen angegeben:

Klasse *Subtree* *(Tree-Root)*

Diese Klasse ergänzt ihre von Tree-Root ererbten Eigenschaften um ein Attribut *parent*, das den Zugriff auf den direkten Vaterknoten eines Teilbaumes erlaubt. Alle Klassen, deren Instanzen Knoten einer Baumstruktur sein können, erben direkt oder indirekt von Subtree.

Die beiden nächstgenannten Klassen bedürfen keiner weiteren Erläuterung:

Klasse *Titled-View-Document* *(Titled-Document, View-Document)*

Klasse *Persistent-Licensed-Object* *(Persitent-Object, Licensed-Object)*

Die nun aufzuführenden Klassen bilden den Ursprung für fast alle Objekte, die in Kapitel 3 identifiziert wurden:

Klasse *IKB-Subtree* *(Subtree, Titled-Document, Persistent-Licensed-Object)*

Alle Strukturen, die innerhalb einer informalen Wissensbasis als Teilbaum in Erscheinung treten können, erben die Subtree-Eigenschaft zusätzlich zu den Eigenschaften der beiden anderen Superklassen. IKB steht für Informal Knowledge Base.

Klasse *IKB-Document* *(Titled-View-Document, Persistent-Licensed-Object)*

Instanzen von Klassen, die von IKB-Document erben, bilden die Hauptbestandteile einer informalen Wissensbasis, also etwa Expertisen- und Domänenmodell, Knowledge-Units, Features, Produktionsklassenhierarchien.

Am Beispiel einer Knowledge-Unit soll die Struktur näher erläutert werden:
Eine Knowledge-Unit kann im Rahmen der Anwendung der KA-Metode in mehrere, formalere Bestandteile dekomponiert werden. Der Bezug zwischen Quell- und Zielunit soll dabei erhalten bleiben. Knowledge-Units können also einen Baum bilden. Außerdem ist eine Unit persitent und mit einer eindeutigen Lizens sowie einem Titel versehen. All diese Eigenschaften erbt sie bereits von der Klasse IKB-Subtree. Die Subtree-Eigenschaft ist auch für die Wurzel-Unit einer Dekompositionshierarchie erforderlich: Sie hat ein Quelltext-Objekt als "Vater". Zusäztlich erben die Units die Eigenschaften eines Text-Document und - über IKB-Document - die eines View-Document zum Zweck ihrer Darstellung am Bildschirm.
Übrigens werden Eigenschaften einer Super-Klasse durch multiple Vererbung nicht etwa

mehrfach an ein- und dieselbe Subklasse vererbt. Der üblicherweise zur Konfliktlösung bei multipler Vererbung eingesetzte Mechanismus ist z.B. in (Keene 89) näher erläutert.

Eine Folge der Anwendung von Vererbungsmechanismen ist die Vereinheitlichung von bei der Analyse vergebenen Namen für Methoden und Slots. Bspw. wird das in der Analyse identifizierte Attribut *Sub-Categories* eines Domänenmodells wie die korrespondierenden Slots in allen übrigen Baumstrukturen schließlich *children* heißen, da es von der Klasse Tree-Root vererbt wird.
Im folgenden Abschnitt wird erläutert, wie die Methode *save* für die Klasse *Text-Document* mittels des Vererbungsmechanismus entwickelt wird.

4.2.3 Entwicklung von Methoden am Beispiel *save*

Entsprechend der in Anhang A.b skizzierten detaillierten Designphase beginnt das Design von Methoden zunächst bei den allgemeinsten Klassen. Die Methode *save* wird daher zuerst für die Klasse *Persistent-Object* definiert, dann für deren Subklassen bis hin zu Feature. Dies geschieht hier unter Verwendung der im Common Lisp Object System (CLOS) gebräuchlichen Notation. Die beteiligten Klassen seien bereits definiert. Ebenfalls definiert seien Zugriffsmethoden in der Form (<slotname> *instance*) für die Slots dieser Klassen. Zum besseren Verständnis wird hier noch die vereinfachte Syntax der defmethod-Form angegeben:

```
defmethod name ( { var | (var class-or-type-name) }* ) {form}*
```

```
(defmethod save ((instance persistent-object) (outstream stream))
       (print (class-name (class-of instance)) outstream))

(defmethod save ((instance persistent-licensed-object) (outstream stream))
       (call-next-method)
       (print (license instance) outstream))

.
.
.
.
(defmethod save ((instance text-document) (outstream stream))
       (call-next-method)
       .
       .
       (print (text instance) outstream))
```

`(call-next-method)` ruft die ererbte Version einer Methode auf. Dadurch wird sichergestellt, daß bei Ausführung einer Methode für eine bestimmte Klasse auch die von Superklassen ererbten Eigenschaften mit berücksichtigt werden. In dem Beispiel etwa werden beim Aufruf der Methode *save* mit einer Instanz der Klasse *Text-Document* und

einem Ausgabestrom als Argument zunächst der Klassenname und die Lizens des Objektes und erst dann der Inhaltstext geschrieben.

5 Prinzipien und Möglichkeiten der Implementation

Nachdem in den Kapiteln drei und vier die Grundzüge der objektorientierten Analyse und des Designs für die integrative Wissensakquisitionsmethode vorgestellt wurden, sollen nun Möglichkeiten zur Umsetzung der Ergebnisse aus diesen Kapiteln in ein funktionierendes Software-System diskutiert werden. Die Ausgestaltung einzelner Methoden und andere Details der Codierung können und sollen dabei nicht erwogen werden; Dies ist den Programmierern einzelner Module zu überlassen. Es sollen hier vor allem Vorgaben und Richtlinien zur Implementation des Systems gegeben werden. Dabei sind folgende Punkte zu berücksichtigen:

- Integration von Fremdsoftware
  Zur Unterstützung der Textanalyse in COKAM+ soll ein auf SUN-Workstations (siehe 5.1.1) laufendes Latent-Semantic-Indexing-Verfahren (LSI) (Dumais et al. 88) zum Zweck effizienten Text-Retrievals der Firma Bellcore nutzbar gemacht werden. Weiterhin ist die Einbindung eines Parsers zur Analyse von textuellen Wissenseinheiten geplant. Dieses Programm ist voraussichtlich ebenfalls auf den Rechnern der Firma SUN verfügbar.

- Portabilität der wesentlichen Teile des Systems
  Der Forschungscharakter des zu implementierenden Systems sollte nicht zum Verzicht auf größtmögliche Portabilität gegenüber anderen Rechnersystemen verleiten. Diese Übertragbarkeit ist nicht zuletzt ein Gradmesser für die Professionalität einer Implementierung. Sie eröffnet auch die Möglichkeit zum späteren Wechsel zu modernerer (schnellerer) Hardware.

- Ausnutzung vorhandener Resourcen
  Vorhandene Hard- und Software-Resourcen sollen soweit sinnvoll und möglich genutzt werden. Dazu gehören neben Rechnern und Software-Tools auch vorhandene Prototypen des geplanten Systems.

- Wiederverwertbarkeit
  Teile des Systems mit eher allgemeinen Funktionen sollten so konzipiert werden, daß sie ohne großen Aufwand in anderen Anwendungen eingesetzt werden können. Dazu gehören etwa eine Schnittstelle zur Inter-Prozeß- und Rechner-Kommunikation oder eine Persistenz-Erweiterung für CLOS.

- Performanz
  Die Akzeptanz des Systems wird wegen seiner stark interaktiven Auslegung u.a. von der Möglichkeit zur verzögerungsfreien Bearbeitung auch großer informaler Wissensbasen abhängen.

- Integration mit anderer Software des ARC-TEC-Projektes
  Ein Ziel des ARC-TEC-Projektes besteht darin, die erstellten Wissenverarbeitungswerkzeuge zu einem Gesamtsystem zu integrieren. Das Wissensakquisitionssystem muß dazu zunächst den technischen Voraussetzungen einer solchen Integration genügen.

- schließlich sollte das System möglichst *rasch* implementiert werden können

Im Zusammenhang mit der Einbindung von Fremdsoftware soll noch das Superbook-System der Firma Bellcore erwähnt werden, das eine sehr komfortabele Oberfläche zum "Browsen" in umfangreichen Texten zur Verfügung stellt. So wünschenswert die Nutzung dieses Systems innerhalb des Wissensakquisitionswerkzeuges sein mag, sie scheitert leider an einer Reihe technischer Probleme, die in angemessenem Zeitaufwand nicht zu bewältigen sind.

Selbstverständlich werden nicht alle oben aufgeführten Punkte optimal zu erfüllen sein. Ein hoher Erfüllungsgrad der erstgenannten Punkte bedingt z.B. einen allzu hohen Zeitaufwand bei der Implementation. Das gleiche gilt in umgekehrter Richtung. Die zu entwickelnden Alternativen sollten daher so formuliert werden, daß man sich nach der Entscheidung für ein einfacheres aber schnelleres Vorgehen nicht von einer stärker befriedigenden Lösung entfernt sondern sich an eine solche annähert. Bevor aber verschiedene Möglichkeiten vorgestellt werden, sollen zunächst die verfügbaren Resourcen betrachtet werden.

5.1 Vorhandene Resourcen

5.1.1 Hardware

Die im ARC-TEC-Projekt installierte Hardware-Umgebung besteht aus zwei SPARC-Server-Maschinen der Firma SUN, einer Reihe von SPARC-SLC-Workstations ohne eigenen Massenspeicher sowie aus einigen Macintosh II Rechnern. Alle diese Maschinen sind über Ethernet in einem lokalen Netzwerk integriert. Die SLC-Workstations werden mit den Server-Maschinen via "Yellow-Pages" und SUN-NFS (Network File System) in einem Cluster gefahren, das ihnen den Zugriff auf die Massenspeichersysteme der Server

sichert. Die Workstations erhalten dadurch die Möglichkeit zu virtueller Adressierung, die wiederum das Fahren von Anwendungen erlaubt, deren Platzbedarf über den Arbeitsspeicher einer einzelnen Workstation hinausgeht. Die Macintosh II-Rechner haben über das AppleShare-System ebenfalls Zugriff auf die Massenspeicher verschiedener Server. Darüberhinaus besteht für sie über die entsprechende TCP/IP-Software voller Zugang zu den in Netzwerken üblichen Diensten wie *remote-login, remote-execution* etc.. Für die Maschinen der Firma SUN, die unter dem UNIX-Betriebssystem laufen, ist die Verfügbarkeit dieser Dienste selbstverständlich. Die Macs verfügen zwar über einen eigenen Massenspeicher, jedoch fehlt ihnen zur Zeit die Möglichkeit zu virtueller Adressierung. Ihre Rechenleistung ist daher bezogen auf die Größe von Anwendungen gegenüber den Suns wesentlich enger begrenzt.

Aktuelle Bemerkung

Eine Besonderheit in der Rechnerausstattung besteht im ARC-TEC-Projekt in der Integration von insgesamt zehn sogenannter Ivory-Boards in die beiden SPARC-Server der Firma SUN. Bei diesen Ivory-Boards handelt es sich um Lisp-Maschinen des Typs GENERA der Firma Symbolics. Sie bestechen durch eine sehr hohe Performanz bei Lisp-Anwendungen sowie durch eine äußerst komfortable Entwicklungsumgebung (siehe auch 5.1.2.1). Als Zielhardware für das in dieser Arbeit diskutierte Wissensaqkuisitionssystem müssen diese Boards in die engste Wahl gezogen werden. Leider standen zum Zeitpunkt der Ausarbeitung nicht genügend Details über ihre Einsatzmöglichkeiten zur Verfügung. Die folgende Diskussion von Implementationsalternativen bemüht sich aber um größtmögliche Offenheit gegenüber solcher Kapazitäten.

5.1.2 Software

5.1.2.1 Software Tools

Es sind hier Programmier-Sprachen und Software-Tools zu besprechen, die auf den oben erwähnten Maschinen verfügbar sind. Dabei wird nur auf diejenigen eingegangen werden, die für die Implementation der Wissensakquisitionsmethode sinnvoll einsetzbar sind. Entscheidendes Kriterium für die Auswahl von Sprachen/Tools zu diesem Zweck ist die Verfügbarkeit einer komfortabel programmierbaren Graphikschnittstelle zur Entwicklung eines Benutzerinterfaces, da hier aufgrund der reichlichen Benutzerinteraktion in COKAM+/CECoS ein Schwerpunkt der Implementationsarbeit liegen dürfte. Eine weitere Einschränkung besteht in einem Übereinkommen innerhalb

des DFKI, hauptsächlich die Programmiersprache LISP als Implementationssprache einzusetzen.

SPRACHEN

Macintosh Allegro Common Lisp

Bei Allegro Lisp für den Macintosh handelt es sich um ein kommerzielles Produkt, das - von kleinen Einschränkungen abgesehen - eine vollständige Common Lisp Implementation im Sinne einer früheren Auflage des Lisp-Standardwerkes von Guy L. Steele (Steele 84) darstellt. Darüberhinaus ist eine Reihe von Erweiterungen enthalten, die teilweise speziell auf die Eigenarten des Apple Macintosh zugeschnitten sind. Dazu gehört eine Object-Lisp genannte Erweiterung, die eine einfache objektorientierte Umgebung realisiert. Object-Lisp beinhaltet alle wesentlichen, in Kapitel zwei genannten Vorzüge von OO-Sprachen, unterliegt aber der Einschränkung, nicht in andere Lisp-Implementationen übertragbar zu sein. Attraktiv wird die Verwendung von Object-Lisp durch die Einbettung fast sämtlicher Mac-typischer Graphik/Window-Fähigkeiten in die objektorientierte Umgebung. Sie ermöglicht eine sehr komfortable und gleichzeitig flexible Gestaltung von Fenstern, Dialogen, Menüs etc., auch unter Ausnutzung der Vererbungsmechanismen. Als weiteres Feature ist die ebenfalls recht komfortable C-Schnittstelle zu nennen, die, wenn es erforderlich scheint, die Einbindung schneller C-Routinen als Lisp-Funktionen sowie den Rückruf von Lisp-Funktionen aus C heraus gestattet. Nicht unerwähnt bleiben sollte schließlich die im typischen Mac-Stil gehaltene, sehr angenehme Programmierumgebung von Mac Allegro Lisp.

Austin Kyoto Common Lisp (AKCL)

AKCL ist eine Public Domain -Implementation vom Common Lisp. Sie enthält wie Mac Allegro Lisp den vollen Sprachumfang aus (Steele 84), ist aber als reine "letter-of-the-law"-Implementation anzusehen. Darin ist nicht unbedingt ein Nachteil zu sehen, da ein AKCL-Programmierer so gezwungen ist, mehr oder weniger voll portablen Common-Lisp-Code zu erzeugen. Eine Besonderheit von AKCL liegt in der zweischrittigen Compilierung von Common Lisp Quellcode: Zunächst wird vom AKCL-Übersetzer C-Code im ANSI-Standard erzeugt, der dann vom jeweils verfügbaren C-Compiler in echten Object-Code übersetzt wird. Dies ermöglicht das Schreiben von C-Routinen innerhalb des Lisp-Quellcodes, wodurch zwar keine sehr komfortable, aber eine brauchbare C-Schnittstelle zur Verfügung steht. AKCL ist bei ARC-TEC auf den Maschinen der Firma SUN verfügbar.

Aktuelle Bemerkung

GENERA LISP

Bei Einsatz der Ivory-Boards steht mit GENERA LISP eines der schnellsten Common Lisps überhaupt in Verbindung mit einer äußerst komfortabelen Entwicklungsumgebung

zur Verfügung. GENERA LISP verfügt auch über ein integriertes Common Lisp Object System (CLOS), das gewisse Nachteile des PCL-Systems aufhebt (siehe unten).

PCL

PCL steht ursprünglich für "Portable Common Loops". Common Loops ist eine objektorientierte Common Lisp Erweiterung, die von der Firma Rank Xerox entwickelt wurde. Mit der Aufnahme des Common Lisp Object Systems (CLOS) in den Common-Lisp Standard wurde PCL auf eine weitgehende Erfüllung der CLOS-Spezifikation hin umgearbeitet. Gegenwärtig erfüllt PCL eine echte Teilmenge der CLOS-Spezifikation, was sich im Gebrauch aber kaum einschränkend bemerkbar macht. PCL ist in Common Lisp geschrieben und daher in fast jede gängige Implementation dieser Sprache portierbar. Die Vorzüge und Möglichkeiten CLOS sind recht umfassend; Sie können hier nicht nicht annährend beschrieben werden. Es sei daher auf (Steele 90) und (Keene 89) verwiesen. Ein Nachteil von PCL besteht in der nicht immer befriedigenden Performanz. Da aber neuere Common Lisp Implementationen gewöhnlich über ein intergiertes (schnelles) CLOS verfügen, ist die Nutzung von PCL nur als Übergangslösung anzusehen. PCL ist als Public Domain Software frei verfügbar.

Als weitere, erwähnenswerte Sprache steht auf allen unter 5.1.1 genannten Maschinen C zur Verfügung. C kann vor allem zur Anbindung von Common Lisp an betriebssystem-nahe Rechner-Resourcen wie etwa Rechner-Kommunikation eingesetzt werden. Eine Anwendung wird in Abschnitt. 5.4 besprochen werden.

GRAPHIK-TOOLS

DFKI-Window-Toolkit

Das 'Window Toolkit' (Becker 90) ist eine am DFKI entwickelte Graphik-Schnittstelle für Lisp zur Programmierung von graphischen Benutzer-Oberflächen. Sein Hauptvorzug besteht darin, sowohl auf den SUN-Rechnern (basierend auf X-Windows) wie auch auf den Macs (via Quickdraw) zur Verfügung zu stehen. Leider realisiert es nur eine Low-Level-Schnittstelle, auf die höhere Funktionen zur einfachen Gestaltung von Dialogen, Menüs etc. nach Art der Graphik-Schnittstelle in Mac Allegro Lisp erst aufgesetzt werden müssen. Ein Minuspunkt ist darin zu sehen, daß zum Zeitpunkt des Entstehens dieser Ausarbeitung das Anzeigen von Pixel-Graphiken in Fenstern nicht oder nur bedingt möglich war. Dieses Feature wird jedoch in dem zu implementierenden System zur Darstellung von Produktmodellen dringend benötigt. Die Frage nach der Nutzung des Toolkits für die hier in Rede stehenden Zwecke wurde daher abschlägig beschieden.

Aktuelle Bemerkung

CLIM

Obwohl das Softwarepaket CLIM (Common Lisp Interface Manager) dem ARC-TEC-Projekt derzeit nicht zur Verfügung steht, wird es in diese Sammlung von Softwareresourcen aufgenommen. CLIM ist eine Common Lisp Erweiterung, die die Gestaltung graphischer Benutzeroberflächen in objektorientierter Weise, basierend auf CLOS, ermöglicht. Die Besonderheit von CLIM besteht darin, daß es sich als ein Standard für die Programmierung von graphischen Benutzeroberflächen in Common Lisp durchzusetzen scheint. Als solcher wird das Softwarepaket über kurz oder lang unabhängig vom Rechnertyp für die meisten kommerziellen Common Lisp Implementierungen verfügbar sein sein. Der Entwicklung von Systemen, deren Portabilität sich auch über die graphische Benutzeroberfläche erstreckt, wird damit kaum mehr ein Problem darstellen.

Interactive Grapher

Der an der Universität Trier entwickelte interaktive Grapher (Waloszek 90) dient der Darstellung von Bäumen und Netzen innerhalb einer graphischen Oberfläche. Er wurde hierzu bereits im COKAM+ -Prototyp eingesetzt und entsprechend spezieller Bedürfnisse verändert und erweitert. Das Programm ist ausschließlich auf Macintosh II Rechnern in Verbindung mit Mac Allegro Common Lisp verfügbar.

5.1.2.2 Frühere Implementation von Teilen des Systems

Als weitere mögliche Resource zur Implementation des Systems kann ein Prototyp gelten, der einen Großteil der COKAM+-Seite der in (Kühn, Schmalhofer, Schmidt 90) geschilderten KA-Methode verwirklicht. Er ist auf MAC-II Rechnern in Allegro Common Lisp unter Verwendung von PCL und dem in 5.1.1 genannten Grapher implementiert und beinhaltet eine Fülle gelöster Teilprobleme, die bei einer Neuimplementation der gesamten KA-Methode in ähnlicher Form wieder auftreten werden. Auf diese kann in Form fertiger Algorithmen zurückgegriffen werden. Als Beispiel seien hier die Verwaltung von Textreferenzen in Knowledge-Units oder die Retrieval-Algorithmen für Knowledge-Units genannt.

Auch der CECoS-Teil der KA-Methode liegt als Prototyp vor. Dieser ist aber in Prolog implementiert und daher nicht direkt in einem Lisp/PCL-System zu nutzen. Die wesentlichen Aspekte der Implementation von CECoS sind dort aber vorgedacht; dies wird die Interagtion in ein Gesamtsystem wesentlich erleichtern.

5.2 Grundlagen aller Alternativen

5.2.1 Aufteilung des Systems

In Kapitel 4.1 wurde bereits die Trennung des Systems in Benutzerschnittstelle und eigentliche Applikation angesprochen. Die Gründe für diese Trennung sollen nun unter Berücksichtigung der zu verwenden Programmier-Sprachen und -Werkzeuge erläutert werden.

In Ermangelung standardisierter Sprachkonstrukte zum Umgang mit graphischen (Bildschirm-) Objekten aller Art als fester Bestandteil von Common Lisp muß zur Schaffung einer graphischen Benutzeroberfläche auf Spracherweiterungen oder Toolkits etc. zurückgegriffen werden, die Rechner- und/oder Betriebssystemabhängig sind und zum Teil den Erwerb zusätzlicher Lizensen mit diversen Weitergabeeinschränkungen erforderlich machen.

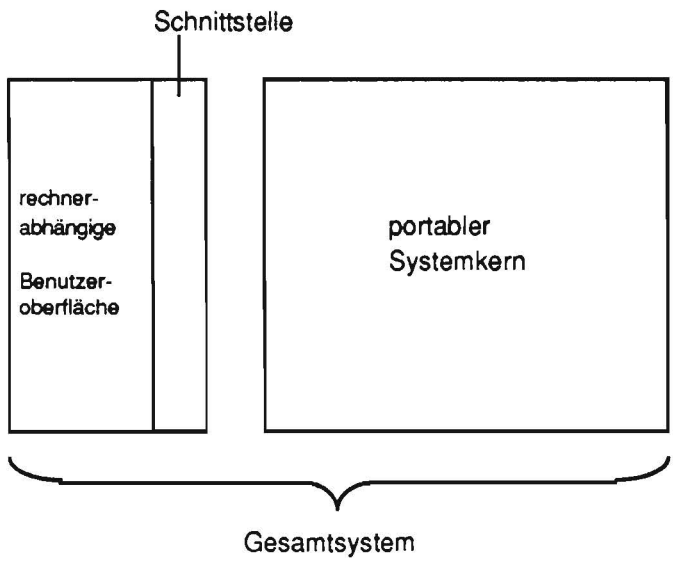

Abbildung 5.2.1.1: Die Aufteilung des Systems in einen rechnerunabhängigen Systemkern und eine rechnerabhängige Benutzeroberfläche. Die Schnittstelle muß für verschiedene Implementationen der Oberfläche gleich bleiben.

Ziel muß daher die Entwicklung einer Architektur sein, die bei einer Portierung des Systems auf ein anderes Rechnersystem nur die Änderung eines eng begrenzten Moduls

erforderlich macht. Der "Kern" des Systems, bestehend aus den in Kapitel 3 und 4 identifizierten und exemplifizierten Klassen und Methoden, sollte daher unabhängig von einer grapischen Benutzeroberfläche formuliert werden. Er darf lediglich Annahmen über eine gleichbleibende Schnittstelle zu dieser Oberfläche enthalten. Abbildung 5.2.1.1 erfaßt diesen Zusammenhang graphisch.

Für diese Trennung spricht auch ein in (Pfaff 85) formulierter Trend: Bei der Entwicklung komplexer und stark interaktiver Systeme sollte die Gestaltung der Benutzeroberfläche nicht mehr dem Anwendungsprogrammierer obliegen, sondern einem *User Interface Programmer*, der mit den ergonomischen Anforderungen an eine Benutzeroberfläche vertraut ist.

Ein weiterer Vorteil dieser Aufteilung liegt in der Flexibilität bei der Implementation des Systemkerns bezüglich der Wahl der Hardware: Es kann aufgrund einer vorläufigen Entscheidung für eine Hardware mit der Implementation des Kerns begonnen werden. Eine Revision dieser Entscheidung ist jederzeit möglich. Voraussetzung ist lediglich die Spezifikation der Schnittstelle zur Benutzeroberfläche.

5.2.2 Prinzipien für die Implementation der graphischen Benutzeroberfläche

Da der interne Aufbau einer Benutzeroberfläche abhängig von der verwendeten Graphiksoftware ist, sollen hier die Mechanismen zur Kommunikation des Systemkerns mit der Oberfläche erörtert werden. Diese Mechanismen sollten für alle zu diskutierenden Implementationsmöglichkeiten gleich sein. Als Richtschnur kann das Modell von CLUE (Common Lisp Userinterface Environment) dienen. CLUE ist eine bei Texas Instruments entwickelte Common Lisp Erweiterung zur Programmierung von Benutzerschnittstellen, die die oben erläuterte Trennung von Oberfläche und Anwendung unterstützt (CLUE 90). Sie beinhaltet eine Reihe von CLOS Klassen, die grundlegende Eigenschaften von Graphikobjekten wie Menüs, Dialogen Fenstern etc. basierend auf X-Windows zur Verfügung stellen. Diese Klassen können durch Vererbung für eigene Zwecke verändert und erweitert werden[2]. Der Mechanismus zur Kommunikation mit der Anwendung ist folgender:

---

[2]Erfahrungen im Umgang mit graphischen Benutzeroberflächen zeigen, daß ein objektorientierter Aufbau gerade für diese Art von Software der weitaus zweckmäßigste ist. Die zur Implementation einer Graphikoberfläche für die KA-Methode in Frage kommenden Tools sollten dieses Merkmal unbedingt aufweisen.

Aufruf von Diensten der Graphikoberfläche durch die Anwendung

Eine Schnittstelle zur Applikation stellt Methoden zur Erzeugung, Manipulation und Disposition von Instanzen der definierten Graphik-Klassen zur Verfügung. Aus Sicht des KA-Systems sollten diese Methoden auch für unterschiedliche Implementationen der Graphikoberfläche stets die gleichen Namen und Aufrufschemata besitzen, also von syntaktischen und semantischen Eigenheiten der Funktionen eines User-Interface-Tools abstrahieren. In Abbildung 4.1.3.1 in Abschnitt 4.3.1 stellen die Dienste *show-pair-comparison-dialog* und *hide-pair-comparison-dialog* zwei Vertreter solcher Schnittstellenmethoden dar. Die Applikation beeinflußt auf diesem Wege das Erscheinungsbild der Graphikoberfläche.

Rückruf von Diensten der Anwendung durch die Graphikoberfläche

Hier geht es um die umgekehrte Richtung: Die Oberfläche übersetzt Aktionen des Benutzers (Maus- und Tastatur-Gebrauch) in Aufrufe von Diensten der Applikation. Jedes Objekt der Graphikoberfläche besitzt dazu eine Anzahl sogenannter **Callbacks**. Diese Callbacks sind mit Diensten der Anwendung assoziiert und werden von Methoden der Graphikobjekte wie Funktionen bzw. Methoden aufgerufen. Sie realisieren auf diese Art etwa die Weitergabe von Benutzer-Anforderungen oder die Übergabe von Werten an die eigentliche Anwendung. Der Grund für die Unterscheidung von Callbacks und Diensten der Anwendung ist folgender: Der Name eines Callbacks gibt die Funktion eines Details des Graphikobjektes wieder, etwa *supply-value* für ein Editierfeld für numerische Werte. Der Name des assoziierten Dienstes hingegen sollte die Bedeutung der Funktion aus Sicht der Anwendung beschreiben, z.B *get-similarity* für den Vergleich zweier Fälle in CECoS. Durch diese Unterscheidung wird die Möglichkeit der unabhängigen Entwicklung von Anwendung und Benutzeroberfläche stärker betont (vergl. CLUE).

5.2.3 Implementation des Systemkerns

Da der Systemkern voll portabel gehalten werden soll, kann auf Einzelheiten seiner Struktur unabhängig von der Diskussion verschiedener Alternativen bei der Hardwareauswahl eingegangen werden. Die Einbindung von Fremdsoftware bleibt dabei zunächst unberücksichtigt, weil die Möglichkeit hierzu in erster Linie von der Wahl der Hardwarekonfiguration abhängt.

Die wesentliche Maßnahme zur Erreichung der Portbilität ist (i) die Wahl einer entsprechenden Programmiersprache sowie (ii) die Definition hinreichend allgemeiner Schnittstellen zu unvermeidbar rechnerabhängigen Teilen des Systems.

(i) Wahl und Gebrauch der Programmiersprache(n)
Im ARC-TEC-Projekt ist die Entscheidung für Common Lisp als Implementationssprache bereits gefallen. Grundsätzlich ist ist die Entwicklung portabler Systeme mit

Common Lisp und CLOS durch die Einführung des ANSI-Standards kein Problem. Dennoch läßt die Definition der Sprache bewußt Lücken, deren Ausfüllung der jeweiligen Implementation überlassen bleibt. Beispielsweise machen eine Unzahl von Common Lisp Standard-Funktionen Gebrauch von optionalen Argumenten, die - bei Nichtangabe durch den Programmierer - mit Defaults belegt werden. Diese Defaults können von einer Implementation zur nächsten variieren und so bei sorgloser Programmierung zu rätselhaften Fehlern bei einer Portierung führen.
Einen weiteren Aspekt liefern die in den meisten Common Lisp-Implementierungen üblichen Erweiterungen des Standards um nützliche Zusatzfunktionen, deren unkontrollierter Gebrauch - obschon er oft verlockend ist - jede Portabilität untergräbt.
Die Abweichungen in der Interpretation des Common Lisp Standards durch verschiedene Implementationen können zumeist durch sorgfältige Programmierung umgangen werden. Kann man auf die Nutzung von Nicht-Standard-Erweiterungen nicht verzicheten, so müssen - wie im Beispiel der graphischen Benutzeroberfläche - die entsprechenden Teile des Systems in separaten Moduln mit angemessenen Schnittstellen untergebracht werden.

(ii) Definition von Schnittstellen
Der COKAM+/CECoS-Systemkern ist, bestehend aus den Definitionen der in Kapitel 3 und 4 identifizierten Klassen praktisch frei von rechnerabhängigen Teilen. Der notwendige Zugriff auf das Filesystem der gewählten Hardware ist durch Standard Common Lisp voll gewährleistet und wird dem System vorrangig über die Klassen *Persistent Object* und *Persistent Object Loader* zur Verfügung gestellt. Daher ist für diese bezüglich der Hardware Alternativen invariante Grundform des Systems lediglich die Spezifikation von Schnittstellenmethoden zur Benutzeroberfläche erforderlich. Welche Methoden das sind und wo diese zum Einsatz kommen, ergibt sich aus den Beschreibungen der Klassen, die von der Klasse *View-Document* erben, siehe Kapitel 3.2, Abschnitt 4.2.2 und Anhang B. Bei der isolierten Implementation des Systemkerns sind diese Methoden zunächst als "Dummys" mit entsprechenden Kommentaren zu definieren. Später liefert dann die Benutzeroberfläche die korrekten Methoden (siehe auch Abschnitt 5.2.2 und 5.3.2.3).

Als weiterer Gesichtspunkt der Kern-Implementation ist eine über das Klassensystem hinausgehende Modularisierung zu erörtern. Zum einen kann die bereits durch Klassendefinitionen gegebene Modularisierung durch Bildung von Lisp-Packages verstärkt werden: Da CLOS die Einschränkung der Sichtbarkeit von Methoden nur unzureichend unterstützt, sollte für jede Klasse ein Package eingeführt werden, aus dem lediglich der Klassenname und die Namen der Dienste - der Methoden die für andere

Klassen sichbar sein sollen - sowie ebensolche Slotnamen exportiert werden. Zum anderen kann das System oberhalb der Klassenebene modularisiert werden: Die in Kapitel 4.1 identifizierten Cluster von Klassen werden als Packages formuliert und exportieren extern benötigte Symbole. Die Implementation dieser Cluster sollte nach den Maßgaben aus Anhang A.c erfolgen.

Die nun zu diskutierenden Alternativen sind nach den bisher getroffenen Entscheidungen in Analyse, Design und Implementation zunächst Alternativen der Hardwareauswahl bzw. der Hardwarekombination. Daraus ergeben sich für die verschiedenen Alternativen Einschränkungen bezüglich der nutzbaren Software, unterschiedliche Möglichkeiten zur Einbindung von Fremdsoftware und insgesamt unterschiedliche Erfüllungsgrade für die eingangs in Kapitel fünf genannten Anforderungen.

5.3 Alternative I - Beschränkung auf Apple Macintosh

Die zuerst zu diskutierende Alternative ist wohl die am einfachsten und schnellsten zu realisierende. Ein Grund dafür liegt in der Möglichkeit, den vorliegenden COKAM+-Prototyp schrittweise in Richtung der in Kapitel drei und vier getroffenen Entscheidungen umzuwandeln und zu erweitern. Ein anderer Grund liegt in der fehlenden Möglichkeit zur Einbindung von Fremdsoftware und dem daraus resultierenden, geringeren Implementationsaufwand. Bei gründlicher Erfüllung der bis hierher beschriebenen Design- und Implementationsvorgaben liefert diese Alternative aber eine Grundlage für die komplexeren Varianten.

5.3.1 Hardware

Die dieser Alternative zugrundeliegende Hardware ist ausschließlich ein beliebiger Rechner der Macintosh II-Familie mit einem Hauptspeicher von 8 MB und einem Prozessor der 68000-Familie. Der Betrieb des Systems wird darauf ohne weitere Kapazitäten wie Netzwerkanbindung etc. möglich sein.

5.3.2 Software

5.3.2.1 Verwendete Sprachen und Tools

Durch die Wahl des Mac II als Hardware-Grundlage ist die zum Einsatz kommende Software vorgegeben. Zur Codierung des Systemkerns in der in Kapitel 5.2 geschilderten Weise wird **Macintosh Allegro Common Lisp** eingesetzt. Es wird dabei nur *reines* Common Lisp nach (Steele 90) verwendet. Auf Basis des Allegro CL

erzeugt man durch Laden des **PCL**-Systems und Bilden eines Lisp-Images eine PCL-Welt, in der dann die Implementatiuon des Systemkerns erfolgen kann.
Für die getrennte Entwicklung der Benutzerschnittstelle wird ebenfalls eine Lisp-Welt erzeugt, die das **Quickdraw**-Package sowie den oben geschilderten, interaktiven **Grapher** enthält. Weitere Software ist zur Implementation dieser Alternative nicht erforderlich.

5.3.2.2 Codierung des Systemkerns

Die in 5.2.3 geschilderten Vorgaben zur Implementation des Kerns werden hier um einige Details erweitert. Das hier gesagte gilt auch für die Implementation des Kerns bei Durchführung der übrigen Alternativen.
Entsprechend der Vererbungshierarchie aus 4.2 und den Vorgaben aus Anhang A.c werden zunächst die allgemeinsten Klassen und nach und nach ihre Subklassen mit sämtlichen Methoden codiert. Für jede Klasse wird ein Package eingeführt, das den gleichen Namen wie die Klasse trägt. Aus diesen Packages werden der Klassenname sowie die Namen von generischen Funktionen (zu diesen zählen auch die in CLOS üblichen Slot-Zugriffs-Funktionen) exportiert. Vor der Verwendung von Diensten einer Klasse bei der Codierung von Methoden anderer Klassen müssen die zu den Diensten gehörenden generischen Funktionen definiert sein. In CLOS bedeuted dies die Festschreibung der Lambdaliste für alle Methoden einer generischen Funktion (siehe Keene 89, Bobrow 88 für Details) und damit eine Vereinbarung für den Gebrauch der generischen Funktion. Dies ist vor allem von Bedeutung, wenn verschiedene Klassen des Systems von verschiedenen Personen codiert werden. Desweiteren hat die Spezifikation der Schnittstellenfunktionen zur Benutzeroberfläche zu erfolgen. Die Definition dieser Funktionen erfolgt in einem User-Interface-Package (siehe 5.3.2.3), wo sie zunächst nur ein Dummy-Verhalten aufzuweisen brauchen. Vorläufig fertige Klassen können nach und nach getestet und durch Bildung übergreifender Packages zu den in Kapitel 4 genannten Clustern zusammengefaßt werden. Zur Unterstützung der Tests von Klassen und Clustern sollte die graphische Benutzerschnittstelle parallel zum Systemkern entwickelt werden.

5.3.2.3 Entwicklung der graphischen Benutzer-Oberfläche

Bisher wurden in 5.2 nur die Prinzipien der Kommunikation zwischen Systemkern und Benutzerschnittstelle erörtert. Im folgenden wird der innere Aufbau in Bezug auf die in dieser Alternative zu verwendende Software besprochen. Die "Object-Lisp" genannte Komponente von Mac Allegro Common Lisp beinhaltet eine Reihe von Standard-Graphik-Objekten als vordefinierte Klassen. Diese Klassen verfügen bereits über die

notwendigen Methoden zur Behandlung von "Ereignissen" (Events) wie Tastenbetätigung oder Mausklicken. Über den Vererbungsmechanismus können diese Methoden um zusätzliches Verhalten erweitert werden. Viele der im Wissensaquisitionssystem benötigten Graphik-Objekte lassen sich ohne weiteres als Instanzen dieser Klassen mit entsprechenden Slotwerten erzeugen. Andere sind einfach mit Hilfe der Vererbung zu konstruieren. Zur Darstellung der Baumstrukturen des Expertise- oder Domänenmodells etwa sind Object-Lisp-Klassen von Fenstern zu definieren, die von der vom interaktiven Grapher bereitgestellten Klasse *Grapher-Window* erben. Tatsächlich brauch sich der Programmierer einer Benutzeroberfläche mittels Object-Lisp nur um *Aussehen* und die assoziierten Funktionen seiner Graphik-Objekte zu kümmern, nicht jedoch um Details der Realisierung dieser Objekte.
Für jede Klasse von Graphik-Objekten, die vom KA-System benötigt werden, wird zweckmäßigerweise ein eigenes Package eingerichtet, in dem das Default-Erscheinungsbild eines Graphik-Objektes, seine Reaktion auf Events etc. definiert werden. Aus dem Package der korrespondierenden Klasse des Systemkerns importiert es außerdem ein Liste der Callbacks, die mit Funktionen des Graphikobjektes assoziiert werden (siehe Kapitel 5.2). Desweiteren erhält jede Instanz eines Graphikobjektes einen Slot, der einen Verweis auf die CLOS-Instanz enthält, der das Graphikobjekt zugeordnet ist. Dies ist notwendig, um die CLOS-Instanz mit Hilfe der Callbacks anzusprechen. Schließlich sind die Schnittstellenfunktionen zu definieren, die einem Objekt des Systemkerns das Erzeugen, Verändern und Entfernen eines zugeordneten Graphikobjektes erlauben. Das Package der entsprechenden CLOS-Klasse importiert deren Namen.

5.3.2.4 Nutzung des COKAM+-Prototyps

Die Erstellung des Wissensaqkuisitionstools auf Basis des Apple Macintosh kann durch weitgehende Ausnutzung des bestehenden Prototyps wesentlich beschleunigt werden. Zu diesem Zweck muß der Prototyp schrittweise in Richtung der bisher formulierten Anforderungen umgearbeitet werden:

(i) Indem man sämtliche Object-Lisp-Codes aus der Anwendung extrahiert und eine Schnittstelle in der oben beschriebenen Art definiert, trennt man die Benutzeroberfläche vom Systemkern. Die Graphikobjekte des Prototyps sind größtenteils separat definiert, ebenso existieren Ansätze zu dem in 5.2.2 besprochenen Kommunikationsmechanismus. Die Trennung dieser Sytemteile muß also lediglich forciert werden. Das Ergebnis dieser Bemühungen wäre dann ein bereits portabler Systemkern.

(ii) Im nächsten Schritt ist ein sukzessiver Umbau des Klassensystem in Richtung der in Kapitel drei und vier entwickelten Strukturen erforderlich. Viele Methoden können dabei beibehalten werden oder müssen zumindest nur geringfügig umgearbeitet werden. Allerdings bedarf das im Prototyp implementierte Ad-hoc-Persistenzsystem einer völligen Neuformulierung im Sinne der Vorgaben aus Kapitel vier.

(iii) Die Durchführung des vorherigen Schrittes wird nicht geringe Mühe kosten; sie liefert aber die einzig tragfähige Grundlage zum Aufbau des Case Experience Combination System (CECoS), das für eine Integration in jedem Fall neu implementiert werden muß, innerhalb der COKAM-Welt. CECoS an einen nicht durch Schritt (ii) bereinigten COKAM-Prototyp anzuhängen, hieße - beinahe - am Augias-Stall anbauen.

Bei mangelnde Vertrautheit mit der Implementation des COKAM-Prototyps ist von einer solchen Umstellung aus zeitlichen Gründen abzusehen, da der Zeitaufwand zur Einarbeitung unangemessen hoch läge. Es sollte dann vielmehr eine Neuimplementation stattfinden, in die Methoden des Prototyps nach gründlicher Prüfung übernommen werden können (siehe auch 5.3.4).

5.3.3 Einbindung von Fremdsoftware

Bedingt durch die Fremdheit der Systeme SUN und Mactintosh sowie durch die beschränkten Kapazitäten des Mac (siehe Kapitel 5.1) ist die Möglichkeit zur Integration der vornehmlich auf SUN WS laufenden Fremdsoftware in dieser Alternative äußerst gering. Jeder Versuch einer Einbindung bedingt zunächst die Portierung der in Rede stehenden Software (LSI, Parser etc, siehe Kap. 5, Anfang) auf den Macintosh. Diese verbietet sich jedoch schon aus reinen Speicherplatzgründen. Darüberhinaus ist es ein absurdes Unterfangen, professionelle Fremdsoftware wie etwa das LSI-Verfahen ohne hinreichende Dokumentation in eine andere Systemwelt zu tragen. Eine Möglichkeit, das LSI-Verfahren und einen Parser bei Verbleib von COKAM+/CECoS auf dem Macintosh dennoch zu nutzen wird als eigene Implementationsmöglichkeit in Kapitel 5.5 vorgestellt.

5.3.4 Bewertung der Alternative

Der oben explizierte Alternative kann hinsichtlich der eingangs in Kapitel 5 genannten Gesichtspunkte eine Beurteilung beigemessen werden. Ein (+) bedeuted positivere, (-) entsprechend eine negativere Bewertung:

- Integration von Fremdsoftware(--)
  Eine Implementation des KA-Systems ausschließlich auf einem Macintosh Rechner muß auf die Unterstützung des Textanalyse durch das LSI-Verfahren ebenso verzichten wie auf den Einsatz eines Parsers zur Formalisierung von Wissenseinheiten. Die Lösung ist in diesem Punkte nicht befriedigend.

- Portabilität der wesentlichen Teile des Systems(+-)
  Da der Systemkern des KA-Tools hardware-unabhängig formuliert werden soll, ist die Forderung nach Portabilität für diesen Teil des Systems in dieser und den folgenden Alternativen erfüllt. Die Portabilität der Benutzeroberfläche ist aufgrund der Verwendung des macintosh-spezifischen Object-Lisp nicht gewährleistet.
- Ausnutzung vorhandener Resourcen(-)
  Von den im ARC-TEC-Projekt vorhandenen Resourcen zur Systemerstellung wird lediglich die Macintosh-Komponente mit der darauf verfügbare Lisp-Entwicklungsumgebung genutzt. Die sich durch die vorhandenen Workstations und und die hochgradige Vernetzung ergebenden Kapazitäten bleiben völlig unberücksichtigt.
- Wiederverwertbarkeit()
  In einem objekt-orientierten System bedeutet Wiederverwertbarkeit von Teilen letztendlich die Wiederverwertbarkeit von Klassenimplementationen. Da Aufbau und Verhalten von zur Wiederverwertung in Frage kommenden Klassen, etwa des Persistenzsystems, oben nur angeregt werden konnten, hängt eine Bewertung dieses Punktes in dieser und den folgenden Alternativen von der praktischen Ausführung ab.
- Performanz(-)
  Erfahrungen mit dem COKAM-Prototyp zeigen, daß die Bearbeitung umfangreicher, informaler Wissensbasen auf dem Macintosh teilweise durch lästige Verzögerungen erschwert wird. Die Gründe hierfür liegen in der beschränkten Rechenkapazität der Macintosh-Rechner (siehe Anhang C) sowie in gravierenden Effizienzmängeln des portablen CLOS-(PCL-) Systems.
- Integration mit anderer Software des ARC-TEC-Projektes(- -)
  Die beschränkte Kapazität der Macintosh-Rechner verbietet leider die softwaretechnische Kombination des KA-Systems mit anderen, größeren Systemen des ARC-TEC-Projektes.
- (+) Der Zeitaufwand zur Erstellung dieser reinen Macintosh-Version des COKAM+/CECoS-Systems kann bei intensiver Nutzung des COKAM-Prototyps auf etwa einen Mann-Monat taxiert werden. Voraussetzung dafür sind Erfahrung im Umgang mit Mac Allegro Lisp und CLOS sowie eine weitreichende Kenntnis der Implementation des Prototyps. Mit nachlassender Erfüllung dieser Voraussetzung dürfte der zu erwartende Zeitaufwand rapide ansteigen (siehe auch 5.3.2.4).

Außerhalb der Bewertung dieser Anforderungen liegt der wesentliche Vorzug dieser Alternative darin, als rasch zu implementierende Grundlage der folgenden, anspruchsvolleren Alternativen dienen zu können.

5.4 Alternative II - Einbeziehung mehrerer Resourcen in einem ansatzweise verteilten System

Die im vorherigen Abschnitt vorgestellte Alternative I ist bezogen auf die zu Anfang des Kapitels 5 formulierten Anforderungen in manchen Punkten unbefriedigend: Die Nutzung ausgefeilter Text-Suchmechanismen in Form des LSI-Verfahrens (Dumais et al. 88) und der Gebrauch eines Natural Language Parsers (Kieras 91) zum Zwecke der Formalisierung von Wissenseinheiten sind fester Bestandteil des Wissensakquisitionskonzeptes in der A-Gruppe des ARC-TEC-Projektes. Diese Punkte bleiben in der ersten Alternative unberücksichtigt. Als Lösung bietet es sich zunächst an das COKAM+/CECoS-System auf die SUN-Maschinen zu portieren. Abschnitt 5.5 diskutiert diesen Ansatz. In der UNIX-Umgebung sind Möglichkeiten zur Nutzung der genannten Software vielfältiger und einfacher nutzbar, etwa durch Bindung der Software zu *einem* Gesamtsystem oder durch Ausnutzung der Techniken der Prozeßkommunikation. Jedoch erscheint eine solche Portierung zur Zeit aus verschiedenen Gründen nicht oder nur bedingt durchführbar (siehe 5.5). Um dennoch zu einem befriedigenden Ergebnis zu gelangen, wird Alternative I nachgebessert, indem man die Nutzung der bei ARC-TEC vorhandenen Resourcen, insbesondere der Netzwerk-Kapazitäten, intensiviert.

Die Motive zur Errichtung eines verteilten Systems liegen häufig in dem Anliegen, spezielle Teilaufgaben eines Softwaresystems auf der jeweils geeignetesten Hardware in einem Rechnerverbund - womöglich parallel - zu bearbeiten. Beim Wissensaquisitions-system des ARC-TEC-Projektes ist die Sachlage ein wenig anders. Verschiedene Softwarepakete, die auf unterschiedlichen Rechnersystemen (Macintosh, SUN) und in unterschiedlichen Sprachen (Lisp, Prolog, C, Fortran) entwickelt wurden, sollen in einem System zusammenwirken. Die verteilte Lösung dieses Problems besteht darin, "ein jedes Ding an seinem Platz" zu lassen und lediglich für die Verständigung zwischen den einzelnen Teilen zu sorgen[3]. Die folgende Definition für ein verteiltes System stammt aus (Sloman 88):

---

[3]Dies ist softwaretechnischer Grund für die Verteilung eines Systems. Gründe für eine Verteilung können selbstverständlich auch in der konzeptuellen Ebene der Anwendung liegen. Ein Beispiel wäre der Wunsch, eine Wissensakquisitionsaufgabe rechnergestützt von verschiedenen Personen an verschiedenen Orten zu verschiedener Zeit zu bearbeiten, ohne die globale Kontrolle über Konsistenz und Suffizienz des erhobenen Wissens zu verlieren.

Ein verteiltes Datenverarbeitungssystenm ist eines, in dem mehrere autonome Prozessoren und Datenspeicher, die Prozesse bzw. Datenzugriffe unterstützen, so kooperierend zusammenarbeiten, daß ein gemeinsames Ziel erreicht wird. Die Prozesse koordinieren ihre Aktivitäten und tauschen Informationen über ein Kommunikationsnetzwerk aus.

Diese Definition intendiert im letzten Satz ein hohes Maß an Autonomie der einzelnen Prozesse. In der hier zu diskutierenden Lösung wird eher eine *Master-Slave*-Beziehung zwischen Prozessen realisiert. Es soll daher von einem *ansatzweise verteilten System* die Rede sein.

5.4.1 Hardware

Neben der bereits in Alternative I zum Einsatz kommenden Hardware, den Macintosh II-Rechnern werden hier auch die zur Verfügung stehenden SUN-Maschinen sowie die der Vernetzung dienende Ethernet-Hardware genutzt.

5.4.2 Software

Die zum Einsatz kommende Software gliedert sich nach der zugrundeliegenden Hardware sowie nach dem genauen Einsatzgebiet:

Macintosh II - Seite

- Für Systemkern und Benutzeroberfläche werden wie in Alternative I
  Mac Allegro Lisp/PCL für den Systemkern,
  Mac Allegro Object Lisp, Interactive Grapher für die Benutzeroberfläche
  eingesetzt.

- Für die Macintosh-Seite des Kommunikationstools finden
  MacTCP zur Netzwerkanbindung,
  MacSockets[4] als Interface zu MacTCP,
  C für den Zugriff auf Ein/Ausgabe-Primitive zur Kommunikation,
  Mac Allegro Lisp/PCL für die Programmierschnittstelle des Tools
  Verwendung.

SUN/UNIX - Seite

- Für die UNIX-Seite des Kommunikationstools wird
  C zur Implementation eines Servers zum Erzeugen neuer Prozesse benützt.

---

[4]MacSockets © 1990 Rijksinstituut voor Visserijonderzoek. All rights reserved.

MacSockets ist eine teilweise Implementation der BSD-UNIX-Sockets des Typs SOCK_STREAM für den Macintosh, basierend auf MacTCP. Es ermöglicht die Entwicklung portabler Rechnerkommunikationsprogramme und wurde für die hier in Rede stehende Anwendung leicht modifiziert.

- Ansonsten finden das LSI-System der Firma Bellcore[5] sowie ein Natural Language Parser Verwendung.

5.4.3 Erweiterung der Alternative I zur Nutzung von Fremdsoftware

Um die in Kapitel 5.3 besprochene Implementationsmöglichkeit zu einem ansatzweise verteilten System auszubauen, sind einige Erweiterungen der Ergebnisse aus den Kapiteln drei und vier vorzunehmen: Zunächst müssen die Recherkommunikationskapazitäten eines in einem Netzwerk eingebundenen Macintosh II-Rechners in Form einer einfach zu bedienenden Schnittstelle in der Lisp/CLOS-Umgebung zur Verfügung gestellt werden. Sodann ist zu überlegen, wie das LSI-Verfahren und der Parser in die COKAM-Welt zu integrieren sind. Auf den Parser muß dabei weniger detailiert eingegangen werden, da zur Kombination dieser Software mit COKAM+/CECoS noch konzeptionelle Arbeit zu leisten ist.

5.4.3.1 Ein Tool zur Rechnerkommunikation für Macintosh Allegro Common Lisp[6]

Die Fähigkeiten eines vernetzten Mac II zur Kommunikation mit anderen Maschinen basieren auf auf einer Macintosh-Version des in lokalen Netzwerken gebräuchlichen TCP/IP-Protokolls (Transmission Controll Protocol/Internet Protocol), MacTCP. Es stellt Primitive zum aktiven und passiven Aufbau von Verbindungen zu anderen Rechnern und zum Senden und Empfangen von Daten zur Verfügung. Basierend auf diesen Primitiven kann in mehreren Schichten ein Kommunikationstool aufgebaut werden, auf das in Macintosh Allegro Lisp/PCL in Form von CLOS-Klassen und -Methoden zugegriffen werden kann. Das Tool soll zunächst folgende Fähigkeiten besitzen:

- Starten und Beenden verschiedener Programme auf verschiedenen UNIX-Rechnern im lokalen Netzwerk bei paralleler Verwaltung mehrerer Verbindungen.
- Kommunikation mit den entfernt gestarteten Programmen über deren Standard-Ein/Ausgabe.
- Möglichkeit zu echt parallelem Rechnen: Ein Lisp-Client auf einem Mac soll nicht auf die Erledigung eines Auftrages durch ein Server-Programm auf einer UNIX-Maschine warten müssen.

---

[5]Die Integration des LSI-Systems in ein zur Anwendung in COKAM geeignetes Paket wurde von Harald Gurres, DFKI Kaiserslautern, vorgenommen.

[6]Im Rahmen dieser Arbeit wird nur eine kurze Beschreibung des Tools vorgenommen; bei hinreichendem Interesse erfolgt eine ausführliche Dokumentation an anderer Stelle.

Für die hier verfolgten Zwecke genügt es, kurz den Aufbau und die Lisp-Schnittstelle des Programmes zu erläutern.

Der Aufbau des Kommunikationstools

Das Tools ist in verschiedenen Schichten realisiert, deren jede die Primitive der nächsttieferliegenden Ebene zu höheren Funktionen integriert und diese teilweise in einer anderen Programmierumgebung zur Verfügung stellt. Abbildung 5.4.3.1.1 veranschaulicht diesen Aufbau graphisch. Die unterste Ebene bildet das oben bereits angesprochene MacTCP-System, das die Standard-Primitive des TCP-Protokolles implementiert. Darauf aufgesetzt ist das MacSocket -System. MacSocket realisiert Teile der auf sogenannten "Sockets" basierenden Rechnerkommunikationsmechanismen des BSD-UNIX-Systems (Sechrest 87) auf dem Mac II. Auf diesem System aufgebaute Programme sind mit geringen Änderungen auch unter UNIX lauffähig.

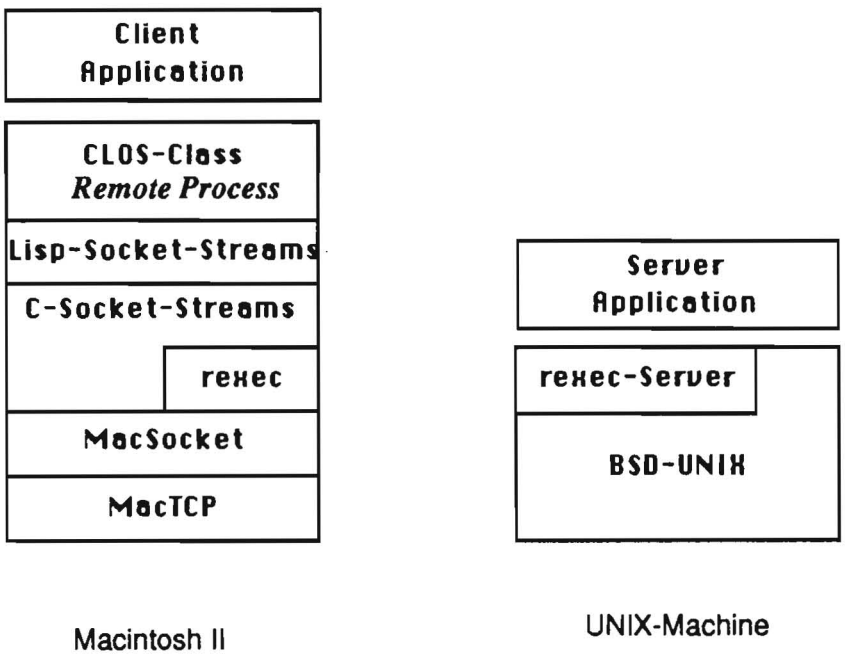

Abbildung 5.4.3.1.1: Die Schichten des Kommunikationstools. Auf der UNIX-Seite ist lediglich die Verfügbarkeit des Remote-Execution-Servers zum Betrieb des Tools sicherzustellen.

Auf der MacSocket-Schicht sind die eigentlichen Teile des Kommunikationswerkzeuges realisiert. Grundlage ist zunächst eine Funktion namens *rexec* (remote execute), die es

ermöglicht, Anwendungen auf einer anderen Maschine zu starten. Sie folgt beinahe exakt dem Protokoll ihres gleichnamigen UNIX-Pendants. Die Ebene *C-Socket-Streams* stellt C-Primitive zur gepufferten Ein/Ausgabe über die von rexec gelieferte Verbindung zur Verfügung. Diese Primitive werden über die C-Schnittstelle von Mac Allegro Lisp zur Implementation spezieller Lisp-Streams in der Schicht *Lisp-Socket-Streams* genutzt. Auf die gleiche Art wird auch der in der *C-Socket-Stream*-Ebene weiter aufbereitete *rexec*-Dienst in Lisp bereitgestellt. Die CLOS-Klasse *Remote-Process* schließlich integriert die nun in Lisp enthaltenen Rechnerkommunikationsmittel zu einer kompakten Einheit.

Die Programmierschnittstelle

Die Dienste des Kommunikationstools werden dem Anwender wie oben angedeuted durch die Klasse Remote-Process zur Verfügung gestellt. Ihre Slots und Methoden sind in Tabelle 5.4.3.1.1 erläutert.

Slot	**Beschreibung**
Remote-ip	Die IP-Adresse einer Unix-Maschine
Remote-port	Die Port-Nummer des remote-Execution-Servers auf der Maschine mit der Adresse *remote-ip*
Program	Das zu startende Programm
Icecream	Ein Eingabestrom, verbunden mit der Standard-Ausgabe des Programmes *program*
Ostream	Ein Ausgabestrom, verbunden mit der Standardeingabe von *program*
Dienst	
close-remote	Schließen der Verbindung
data-arrived-p	Das Prädikat gibt an, ob zur Zeit Daten von *istream* gelesen werden können
rcall	Übermittlung eines Auftrags an *program*

Tabelle 5.4.3.1.1: Die wesentlichen Slots und Methoden der CLOS-Klasse *Remote-Process*

Beim Erzeugen einer Instanz der Klasse *Remote-Process* werden über Schlüsselworte die Slots *remote-ip*, *remote-port* und *program* belegt. Sind die Angaben sinnvoll, wird die Verbindung aufgebaut und die erzeugt Instanz an den Anfang einer Liste `*remote-processes*` gehängt. Ansonsten erfolgt eine Fehlermeldung. Die normalen Common

Lisp Ausgabefunktionen können nun nach erfolgreicher Verbindungsaufnahme Daten an den entfernten Prozeß über den Strom *ostream* übergeben. Das gleiche gilt umgekehrt für den Eingabestrom *istream*, mit der Einschränkung, daß die Auswertung der generischen Funktion `data-arrived-p` zuvor "T" liefern muß. Die generische Funktion `rcall` schickt Aufträgen an den entfernten Prozeß und liest die Ergebnisse. Sie soll etwas genauer betrachtet werden:

rcall *message instance* &key :read-function :non-blocking :timeout-value

instance muß eine Instanz der Klasse Remote-Process sein. Die *message* wird in ihrer *printed-representation* an den mit *instance* verbundenen Prozeß gesandt. Das weitere Verhalten ist von den Werten der Schlüsselworte abhängig: Eine zu `:read-function` angegebene Funktion wird zum Lesen des Ergebnisses der Anfrage vom Eingabestrom *istream* der Instanz genutzt. Der Default für `:read-function` ist die Lisp-Funktion `read`. Erhält das Schlüsselwort `:non-blocking` den Wert `nil` (die Defaulteinstellung), wartet der Aufruf von `rcall` bis zum Wert von `:timeout-value` auf das Ergebnis. Der Wert des Aufrufs von `rcall` ist dann der Wert der auf den Eingabestrom von *instance* angewandten `read-function`. Trifft das Ergebnis nicht rechtzeitig ein, erfolgt eine Fehlermeldung. Wird das Schlüsselwort `:non-blocking` mit einem Non-nil-Wert (etwa T) angegeben, so terminiert die Auswertung von `rcall` ohne Umschweife; als Wert wird ein *Closure*[7] zurückgegeben, das an ein Symbol, etwa *foo* gebunden werden kann. Zu einem beliebigen, späteren[8] Zeitpunkt liefert dann der Aufruf `(funcall foo)` das Ergebnis der Anfrage an den entfernten Prozeß. Auf diese Weise kann echte Parallelverarbeitung realisiert werden.

Wird ein Remote-Process nicht mehr benötigt, so muß er mit `(close-remote instance)` geschlossen werden, bevor die entsprechende CLOS-Instanz "garbage-collected" wird. Es können maximal 64 Instanzen der Klasse *Remote-Process* gleichzeitig mit entsprechend vielen Prozessen auf verschiedenen Rechnern verbunden sein.

---

[7] In diesem Kontext ist ein *Closure* eine zur Laufzeit erzeugte Lisp-Funktion, in der die zum Zeitpunkt der Erzeugung bestehenden, lexikalischen Bindungen gleichsam "eingefroren" sind.

[8] "Später" bedeuted hier: Frühestens dann, wenn ein Ergebnis eingetroffen ist. Die generische Funktion `data-arrived-p` stellt dies fest. Ein verfrühter Aufruf des Closures liefert eine Fehlermeldung (ohne einen erneuten Versuch auszuschließen).

Anforderungen an aufrufbare Programme

Ein Programm, das mit Hilfe des Kommunikationswerkzeuges aufgerufen werden soll, unterliegt derzeit folgender Einschränkung: Es muß seine Eingaben vom Standardeingabe-Strom lesen und seine Ausgaben auf die Standardausgabe zurückliefern. Geeignet sind daher besonders Systeme, die einen *Command-Interpreter* zur Verfügung stellen, wie etwa Lisp, Prolog oder eine Mail-Utility. Ein Vorteil dieser Einschränkung liegt darin, daß ein aufzurufendes Programm nichts von seiner Verwendung durch einen entfernten Rechner zu "wissen" braucht, d.h. es muß nicht extra zu diesem Zweck modifiziert werden.

5.4.3.2 Nutzung der LSI-Software

Bevor die Einbindung des LSI-Verfahrens in das COKAM+/CECoS-System diskutiert wird, soll das Verfahren selbst kurz erläutert werden:

LSI

Das LSI (Latent Semantic Indexing)-Verfahren wurde bei BELLCORE (Dumais et al. 88) zum dem Zweck entwickelt, aus einem Text einzelne Textabschnitte herauszufiltern, die zu einer gegebenen Anfrage in inhaltlichem Bezug stehen. Die Größe eines Textes kann von einige Seiten bis hin zu Buchumfang variieren. Ein Textabschnitt soll aus einem oder mehreren Sätzen bestehen dürfen. Eine Anfrage kann sich aus Schlüsselworten oder schon gegebenen Textabschnitten zusammensetzten, die aber jeweils im Gesamttext enthalten sein müssen. Die Ausgabe des Verfahrens besteht in einer Folge von Abschnitten, die nach ihrer "Ähnlichkeit" zur Anfrage absteigend geordnet ist. Das Verfahren arbeitet in etwa wie folgt: Ein Gesamttext muß, vollständig in einzelne Abschnitte, etwa Sätze, zergliedert, vorliegen. Das Verfahren sammelt Schlüsselworte des Textes und bildet aus diesen und den Abschnitten eine Matrix, die das Vorkommen von Worten in Abschnitten notiert. Diese singuläre Matrix wird mit einer Singuläre-Werte-Dekomposition in drei kleinere, reguläre Matrizen umgerechnet. Auf diese Weise wird jedes Wort und jeder Abschnitt des Textes mit einem Punkt in einem Vektorraum assoziiert. Das Skalarprodukt zweier Vektoren dieses Raumes liefert dann ein Ähnlichkeitsmaß für die ihnen zugeordneten Textobjekte (Schlüsselworte oder Abschnitte). Die Verteilung der Textobjekt-Vektoren im Vektorraum soll so eine "latente Semantik" des Textes widerspiegeln. Nähere Einzelheiten finden sich in (Dumais et al. 88).

Einbindung in COKAM+ auf dem Macintosh

In COKAM+ soll das LSI-Verfahren einen Experten bei der Suche nach Textstellen aus einem Quelltext unterstützen, die bezüglich der zu bearbeitenden Problemstellung relevant

erscheinen und daher in Knowledge-Units kopiert werden sollen. Um das auf SUN-Rechnern lauffähige LSI-System in COKAM+ verfügbar zu machen, wird das oben beschriebene Kommunikationstool eingesetzt.

Ein einfaches Kommunikationsprotokoll

Es ist zunächst ein Protokoll festzulegen, das die Kommunikation zwischen COKAM+ und LSI regelt. Dieses besteht im wesentlichen aus drei Kommandos, die von einer Art LSI-Command-Interpreter von der Standardeingabe gelesen und in entsprechende Aufrufe des LSI-Systems übersetzt werden:

- prepare-text
  Dieses Kommando veranlaßt LSI, einen Text von der Standardeingabe zu lesen bis eine Ende-Marke erreicht wird. Der Text wird daraufhin für die LSI-Suche prepariert und in einer internen Darstellung abgelegt. Eine Statusmeldung auf die Standardausgabe schließt die Aktion ab.

- recognize
  Auf dieses Kommando folgend, wird der Name eines Textes übergeben. Die LSI-Shell stellt fest, ob der angegebene Text bereits zur LSI-Suche prepariert wurde und gibt eine entsprechende Rückmeldung.

- query
  Dem Kommando *query* folgen Worte und Textabschnitte, die als LSI-Anfrage interpretiert werden. Das zurückgelieferte Ergebnis ist eine Liste der Anfangs- und Endemarken der gefunden Textabschnitte bezogen auf den Originaltext, jeweils assoziiert mit einer Ähnlichkeitsrate.

Die Übermittlung eines Originaltextes an das LSI-System bringt eine redundante Datenhaltung mit sich; diese birgt aber keine Nachteile, da es in COKAM+ ohnehin verboten ist einen Quelltext zu ändern und daher keine Konistenzprobleme zu befürchten sind.

Integration in die COKAM+/CECoS Klassenhierarchie

Zur Bereitstellung der LSI-Kommandos in COKAM+ wird eine CLOS-Klasse *LSI-Process* gebildet, die direkt von *Remote-Process* erbt. Als Subklasse der Klasse *LSI-Process* sowie der Klasse *Sourcetext* erzeugt man dann eine Klasse *LSI-Sourcetext*, die die bisherigen Eigenschaften eines Quelltextes mit denen eines *LSI-Process* integriert. Die Methoden der Klasse *LSI-Process* entsprechen den Kommandos der LSI-Shell auf der SUN-Seite. Jede Instanz der Klasse *LSI-Sourcetext* ist dann - unter anderem - eine virtuelle Ausgabe eines in Wahrheit auf einer Sun-Workstation laufenden LSI-Prozesses. Wird eine solche Instanz erzeugt, erfolgt - als Erbe der Klasse *Remote-Process* -

automatisch die Verbindungsaufnahme mit einem LSI-Prozeß auf einer SUN-Workstation. Die hierzu erforderliche Adressinformation (siehe 5.4.3.1) sollte, um eine hohe Transparenz der Netzwerknutzung zu erreichen, in Form von Defaultwerten zur Verfügung stehen. Nur wenn ein Rexec-Server unter der Default-Adresse nicht ansprechbar ist, sollte der Benutzer nach den korrekten Angaben befragt werden.

Slots	Beschreibung
Default-ip (class)	Die IP-Adresse eines Rechners, auf dem ein Rexec-Server läuft.
Default-port (class)	Die Port -Nummer des Rexec-Servers auf dem Rechner mit der IP-Adresse *default-ip*
Lsi-status	Enthält Status-Information über das LSI-System bezogen auf den aktuellen Quelltext (siehe Anhang D.a)
Status-function	Die *status-function* ist an geeigneter Stelle durch das Event-System aufzurufen, um den Benutzer über den Zustand des LSI-Systems zu informieren. Um dynamisch und leichter austauschbar zu sein, wird die Funktion in einem Slot gehalten (siehe Anhang D.a).
Methoden	
get-new-address	Diese Methode erfragt eine gültige Server-Adresse beim Benutzer.
prepare-text	Ein anzugebende Textdatei wird für die LSI-Suche vorbereitet
query	Mit der Methode *query* wird eine LSI-Anfrage gestellt.
recognize	*recognize* stellt fest, ob ein Text bereits für die LSI-Suche prepariert wurde.

Tabelle 5.4.3.2.1: Die Slots und Dienste der Klasse LSI-Process (ohne die ererbten Attribute). Mit (class) markierte Slots sind der Klasse, nicht einer Instanz, zugewiesen, d.h. sie haben für alle Instanzen der Klasse stets den gleichen Wert. Um die genannten Dienste zu implementieren, können weitere Methoden erforderlich sein.

Die Methoden *prepare-text*, *recognize* und *query* der Klasse *LSI-Process* könne mit Hilfe der von der Klasse *Remote-Process* ererbten Methoden entwickelt werden. Ein Vorschlag zu ihrer Codierung findet sich in Anhang D.

Die Art der Einbindung des LSI-Verfahrens durch Vererbung der *Remote-Process*-Eigenschaften auf die Klasse *LSI-Sourcetext* bedingt, das für mehrere, gleichzeitig bearbeitete Quelltexte je ein neuer LSI-Prozess auf einer entfernten Maschine gestartet wird. Da in dem UNIX-Prozess eines *LSI-Sourcetexts* ja der inhaltlich gleiche Text bearbeitet wird wie in der CLOS-Instanz selbst, ist diese Integration von Prozess und Instanz durch Vererbung sinnvoll[9]. Wegen der geringen zu erwartenden Zahl gleichzeitig zu bearbeitender Quelltexte ist sie auch ohne weiteres durchführbar.

Um die Methoden eines *LSI-Sourcetexts* anzusprechen, definiert man innerhalb der graphischen Benutzerschnittstelle geeignete Dialogobjekte, die es einem Benutzer erlauben, LSI-Anfragen zu formulieren und ihre Ergebnisse auszuwerten. Darauf soll hier nicht weiter eingegangen werden.

5.4.3.3 Nutzung eines Parsers

Der in COKAM+ einzusetzende Parser (Kieras 91) zur Formalisierung von Wissenseinheiten ist aller Voraussicht nach als Lispfunktion implementiert und steht daher im Rahmen eines *Lisp-Toplevels* zur Verfügung. Er erfüllt damit die Anforderung zum Einsatz mit Hilfe des Kommunikationswerkzeuges. Die Einbindung erfolgt genau wie beim LSI-System, indem man eine Unterklasse der Klasse *Remote-Process* erzeugt, die die Funktionalität des entfernten Programmes durch geeignete Methoden virtualisiert. Diese Klasse, nennen wir sie *Parser-Process*, vererbt ihre Eigenschaften jedoch nicht etwa an die Klasse *Knowledge-Unit*. Stattdessen wird zur Laufzeit des COKAM+/CECoS-Systems nur eine Instanz der Klasse *Parser-Process* erzeugt, auf deren Dienste in einigen Packages zugegriffen werden kann. Eine Integration nach dem Vorbild der Klasse *LSI-Sourcetext* macht hier keinen Sinn: Eine Wissenseinheit bedarf weder einer besonderen Preparation auf der Seite des entfernten Prozesses noch einer entsprechenden, doppelten Datenverwaltung, um dem Parser zugänglich zu sein. Der wesentliche Dienst der Klasse *Parser-Process* besteht in der Methode *parse*; sie wird mit einer Wissenseinheit als Argument (etwa der *current-unit* des Objektes *Unit-Stack*) gesteuert durch ein Dialogobjekt aufgerufen. Die Klasse *Parser-Process* enspricht ansonsten in etwa der Klasse *LSI-Process* und soll daher nicht weiter erlätert werden.

---

[9]Ganz abgesehen von einer gewissen Eleganz.

5.4.4 Bewertung der Alternative

Ebenso wie der Alternative I soll dieser erweiterten Implementationsmöglichkeit eine Bewertung zugemessen werden:

- Integration von Fremdsoftware (++)
  Indem man die Rechnerkommunikationsmittel des Macintosh II Rechners ausnutzt, ist es nun möglich, das LSI-System und den Parser innerhalb des COKAM+-/CECoS-Systems zur Verfügung zu stellen. Die Lösung ist insofern recht befriedigend.
- Portabilität der wesentlichen Teile des Systems(+)
  Für die Portabilität gilt im wesentlichen das für die Alternative I gesagte. Das Kommunikationstool ist in den in C programmierten Teilen und den Common Lisp-Teilen beinahe voll UNIX-kompatibel.
- Ausnutzung vorhandener Resourcen (++)
  Neben der Macintosh-Komponente mit der darauf verfügbare Lisp-Entwicklungsumgebung werden nun auch weitere Soft- und Hardwareresourcen genutzt: Die Kommunikationssoftware auf SUN- und Mac-Rechnern sowie die SUN-Rechner selbst mit zugehörigen Vernetzung gehören dazu.
- Wiederverwertbarkeit (+)
  Das Kommunikationstool ist unabhängig von der Implementation des Wissensakquisitionssystems in allen Mac Allegro Lisp Anwendungen einsetzbar und kann überdies auf andere Systeme (SUN, SYMBOLICS) portiert werden.
- Performanz (-)
  Für die Performanz gilt das bereits zu Alternative I gesagte. Die Verteilung bringt in dieser Hinsicht keine Vorteile (aber auch keine Nachteile!), da die wesentlichen Teile des Systems nach wie vor auf einem MAC II laufen. Die Teile der Funktionalität des Systems, die erst durch die Verteilung erschlossen werden (LSI), laufen bedingt durch die Paralellverarbeitung sicherlich schneller, als eine vergleichbare lokale Realisierung.
- Integration mit anderer Software des ARC-TEC-Projektes(+)
  Das Kommunikationswerkzeug eröffnet die Möglichkeit, mit einer ganzen Reihe andere Programme auf entfernten Maschinen zu kommunizieren. Die sich daraus ergebende Verteilung eines ARC-TEC-Gesamtsystems ist vielleicht die modernste und vielversprechendste Möglichkeit zur Integration.
- (+)Setzt man das Kommunikationstool und eine Implementation der Alternative I als vorhanden voraus, so kann die Einbindung des LSI-Systems in wenigen Mann-

Tagen bis Mann-Wochen erfolgen. Auch die Einbindung des Parsers dürfte wenig Probleme bereiten, jedoch fehlen noch konzeptionelle Grundlagen.

5.5 Alternative III - Portierung der Alternative II auf Sun/Unix

Hier soll die Implementation des Wissensakquisitionssystems in einer reinen UNIX-Umgebung, auf dem SPARC-System der Firma SUN, besprochen werden. Diese Alternative entfernt sich am weitesten von der Implementation des COKAM+-Prototyps. Bedingt durch die Kapazitäten des UNIX-Betriebssystems in Verbindung mit der leistungsfähigen Hardware der SUN-Maschinen stehen weitreichende Möglichkeiten zur Nutzung von Fremdsoftware und der Kommunikation mit Programmen auf anderen Maschinen zur Verfügung. Eine Einschränkung bei der Umsetzung dieser Alternative erwächst aus dem derzeitigen Mangel an geeigneter High-Level-Graphiksoftware zur Entwicklung der Benutzerschnittstelle (siehe auch 5.4.2.1).

5.5.1 Hardware

Die Hardware, die die maschinelle Grundlage dieser Alternative darstellt, besteht - wie oben angedeutet - aus Rechnern der SPARC-Serie der Firma SUN. Dabei ist es gleichgültig, ob eine SPARC-SLC-Workstation ohne eigenen Massenspeicher oder eine der Server-Maschinen zu Entwicklung und Einsatz des Systems benützt werden. Die erstellte Software kann später ohne Mühe auf andere Rechner-Typen, die unter dem BSD-UNIX-System laufen, portiert werden. Die SPARC-Rechner sind mit einem neuen Risc-Prozessor ausgestattet, der gegenüber den in herkömmlichen SUN-Maschinen und den Macintosh-II-Rechenern installierten 68000-Prozessoren eine deutlich erhöhte Verarbeitungeschwindigkeit aufweist (siehe Anhang C).

5.5.2 Software

5.5.2.1 Verwendete Sprachen und Tools

Die auf den SUN-Computern zur Verfügung stehenden Lisp-Implementation ist AKCL (Austin Kyoto Common Lisp). In Verbindung mit dem PCL-System erhält man eine CLOS-Welt, die mit kleinen Einschränkungen die in (Steele 90) gestellten Anforderungen erfüllt. Innerhalb dieser Welt wird der Kern des Wissensakquisitionssystems erstellt.
Welche Software zur Entwicklung der graphischen Benutzerschnittestelle auf den SPARCs eingesetzt werden soll, ist zur Zeit unklar; das DFKI-Window-Toolkit scheidet aus den in 5.1.2 genannten Gründen aus. Andere Toolkits stehen dem ARC-TEC-Projekt

derzeit nicht zur Verfügung. Geeignet erscheinen etwa CLUE und CLIM, von denen das letztere möglicherweise angeschafft werden wird. In Abhängigkeit der Entscheidung für ein Window-Toolkit muß zusätzlich ein Grapher zur Darstellung von Baum/Netz-Strukturen entwickelt oder anderweitig beschafft werden.

5.5.2.2 Codierung des Systemkerns

Da der Systemkern portabel ausgelegt ist, unterscheidet sich die Codierung in der UNIX/AKCL/PCL-Umgebung abgesehen von Merkmalen der Entwicklungsumgebung durch nichts von dem unter 5.2.3 und 5.3.2.2 beschriebenen Vorgehen. Im günstigsten Fall steht bereits eine fertige Implementation nach Alternative I zur Verfügung. Der dort erstellte Systemkern muß beinahe ohne Modifikationen in der AKCL-Umgebung lauffähig sein.

5.5.2.3 Entwicklung der graphischen Benutzer-Oberfläche

Da eine Entscheidung für ein geeignetes Window-Toolkit zur Entwicklung graphischer Benutzeroberflächen auf SUN-Rechnern in Lisp noch aussteht, müssen detailierte Vorschläge zur Durchführung dieses Schrittes für das Wissensakquisitionssystem leider unterbleiben. Es wird aber auf Abschnitt 5.2.2 verwiesen, in dem Prinzipien für die Enwicklung einer graphischen Oberfläche aufgeführt werden.

5.5.3 Einbindung von Fremdsoftware

Für die Nutzung von nicht oder nur teilweise bei ARC-TEC entwickelter Software ergeben sich in der UNIX-Umgebung vielversprechende Möglichkeiten. Diese basieren auf der Kapazität der SUN-Systeme: Es bestehen dort verschiedene Möglichkeiten zur Prozeß- und Rechnerkommunikation, die mit Hilfe der Programmiersprache C über entsprechende Schnittstellen relativ leicht in Lisp-Systeme integriert werden können. Die einfachste und komfortabelste Möglichkeit, Fremdsoftware für das Wissensakquisitionssystem in einer UNIX-Umgebung zu nutzen, besteht darin, das für den Macintosh entwickelte Kommunikationstool nach AKCL (Austin Kyoto Common Lisp) unter Unix zu portieren. Es kann dann ebenfalls parallel auf verschiedenen Maschinen oder wahlweise auch auf nur einer Maschine - in verschiedenen Prozessen - gerechnet werden. Die Maßnahmen zur Integration der Fremdsoftware in das Wissensakquisitionssystem unter UNIX sind dann exakt die gleichen wie die in Alternative II unter 5.4.3.2 und 5.4.3.3 besprochenen: Die Programmierschnittstelle des Kommunikations-Werkzeuges - die Klasse *Remote-Process* - bleibt bei der Portierung unverändert.

Portierung des Kommunikationstools nach AKCL

Das "Herzstück" des Kommunikationstools besteht hauptsächlich in der *C-Socket-Streams*-Ebene aus Abbildung 5.4.3.1.1. Wie in Abschnitt 5.4.3.1 erläutert wurde, ist dieser Teil des Tools - von geringen Einschränkungen abgesehen -. voll UNIX-kompatibel. Die MacSocket- und MacTCP-Werkzeuge sind unter UNIX überflüssig, da die von ihnen bereitgestellte Funktionalität bereits im UNIX-System integriert ist. Lediglich die *Lisp-Socket-Streams*-Ebene des Kommunikationstools ist für eine Portierung nach UNIX neu zu implementieren, da sie Gebrauch von einer C-Schnittstelle macht. Die ist aber in jeder Common-Lisp-Implementation unterschiedlich. Abbildung 5.5.3.1.1 veranschaulicht den Aufbau des Kommunikationstools unter UNIX.

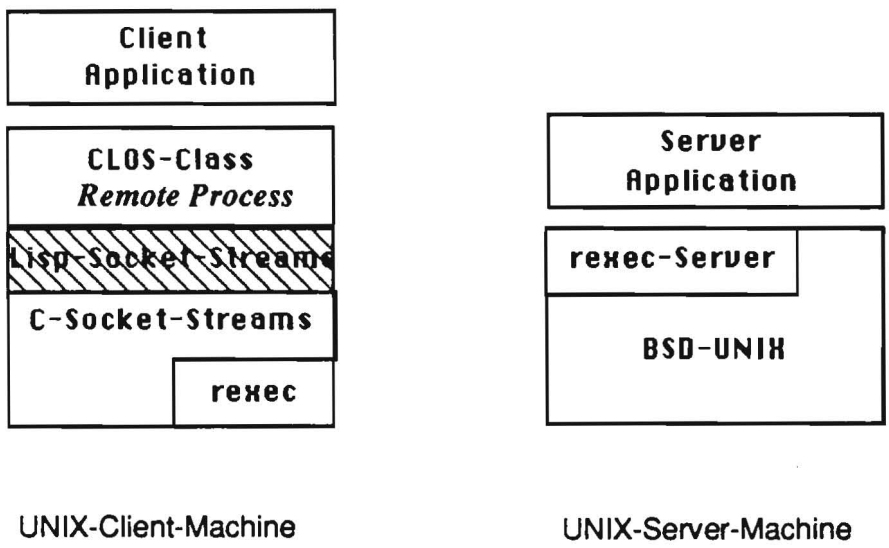

Abbildung 5.5.3.1.1 Die UNIX-Version des in 5.4.3.1 dargestellten Kommunikationstools. Lediglich die in der Graphik durch Schraffur gekennzeichnete Ebene muß in AKCL reimplementiert werden. Bei der Server- und der Client-Maschine kann es sich auch um ein und dieselbe Maschine handeln.

5.5.4 Bewertung der Alternative

Diese Alternative schneidet bei einer Bewertung bezogen auf bereits bekannten Kriterien um einiges besser ab, als die vorherigen beiden:

- Integration von Fremdsoftware (++)
  Durch die Portierung des Kommunikationstools ergeben sich die guten Möglichkeiten zur Nutzung des LSI-Systems und des Parsers auch für diese Alternative
- Portabilität der wesentlichen Teile des Systems(+)
  Wird für die Implementation der Benutzerschnittstelle ein portables Werkzeug (etwa CLIM) eingesetzt, so läßt die UNIX-Alternative bezüglich dieses Kriteriums kaum Wünsche offen.
- Ausnutzung vorhandener Resourcen (++)
  Die oben besprochene Alternative verabschiedet sich vollständig von der Nutzung der Macintosh-Rechner, worin aber kein Nachteil zu erblicken ist. Die Portabilität zu diesen Maschinen bleibt ja erhalten. Enscheidend ist, das die größeren Kapazitäten genutzt werden. Die liegen in den vernetzten SUN-Rechnern
- Wiederverwertbarkeit (+)
  An wiederverwertbaren Teilen des Sytsems ist hier das - nunmehr portierte - Kommunikationstool zu nennen.
- Performanz (+)
  Durch den Verzicht auf die Macintosh-Computer wird ein deutlicher Gewinn an Verarbeitungsgeschwindigkeit zu verzeichnen sein (vergl. Anhang A.c) Darüberhinaus kann mit bei weitem größeren Wissensbasen gearbeitet werden.
- Integration mit anderer Software des ARC-TEC-Projektes(++)
  Die in der Bewertung der Alternative II genannten Möglichkeiten zur Integration gelten in vollem Umfang auch für diese Alternative; darüberhinaus erlaubt die Kapazität der SUN-Rechner auch eine Verschmelzung ganzer Programmsysteme zu einem System auf einer Maschine.
- (-)Ein Minuspunkt für diese Alternative liegt in der zu erwartenden Entwicklungsdauer: Während die Portierung des Systemkerns und des Kommunikationswerkzeuges in zwei bis drei Mann-Wochen zu bewerkstelligen sein sollte, wird der Aufwand für die Neuentwicklung der Benutzeroberfläche mit einem noch unbekannten Werkzeug nur in Mann-Monaten zu messen sein. Eine genauere Schätzung käme über die Zuverlässigkeit eines Horoskopes nur unwesentlich hinaus.

Aktuelle Bemerkung

In der Weise, in der eine Portierung des Wissensakquisitionssystems auf SUN-Rechner vonstatten gehen kann, sollte sie mittlerweile auch für Symbolics-Ivory-Boards möglich sein. Da die Boards über eine Netzwerkanbindung und die Fähigkeit zu *Remote-Procedure-Calls* verfügen, sollte auch die Portierung des Kommunikationswerkzeuges ohne weiters machbar sein. Desweiteren steht auf den Symbolics-Boards nun CLIM zur Verfügung. Der Entwicklung eines voll potabelen COKAM+/CECoS-Systems in dieser Umgebung steht damit - abgesehen vom zu erwartenden Arbeitsaufwand - nichts mehr im Wege.

Anhang

A Der objektorientierte *Systems Life Cycle*

Die folgende Beschreibung eines Life-Cycle-Modelles für objektorientierte Systeme faßt die Grundzüge einiger erwähnten Methoden in knapper Form zusammen. Sie gründet sich auf eine Arbeit von Henderson-Sellers und Edwards (Henderson-Sellers, Edwards 90) und auf Anregungen aus (Edward, King, Winblad 90).
Genau wie im traditionellen Life Cycle Model lassen sich grob drei Phasen der Entwicklung identifizieren:

- objektorientierte Analyse
- oo. Design
- oo Implementation & Verifikation

Die Phasen umfassen eine Reihe von Einzelschritten, die sich überlappen und bei Bedarf iteriert werden können. Die Übergänge zwischen den einzelnen Phasen sind in der objektorientierten Entwicklung fließend und die Abgrenzung ist daher willkürlich. In der Literatur herrscht weder Konsens über diese Einteilung noch über den Inhalt einzelner Schritte dieser Phasen.

A.a Objektorientierte Analyse

Bei den Methoden der SD ist es das Anliegen der User Requirements Analysis and Specification, die Funktionen des geplanten Systems in einer dem Benutzer verständlichen Terminologie zu beschreiben. Im Gegensatz dazu versucht die objektorientierte Analyse, die *Objekte* der Problemdomäne zu identifizieren. Den Ausgangspunkt für die Analyse kann eine textuelle Beschreibung des Problemes bzw. der Problemdomäne sein. Das Interesse gilt dabei zunächst nur den Entitäten, die dem intendierten Anwender bei seiner Sicht auf das Problem bewußt sind. Die resultierende Beschreibung einzelner Objekte umfaßt ebenfalls nur Slots und Dienste, deren Existenz einem Anwender bei einer äußeren Perspektive evident ist, also die Schnittstellen der Objekte. Diese Schnittstellen machen teilweise zu einem späteren Zeitpunkt das für den Anwender sichtbare Verhalten des Systems aus.
Der Problemraum wird bei dieser Vorgehensweise hinreichend für die Problemlösung, jedoch nicht mehr als notwendig modelliert. Das Ergebnis ist ein vorläufiges Objektwörterbuch, das als objektorientierte User Requirements Specification angesehen

werden kann. Über die Implemenation von Diensten wird dabei keinerlei Aussage gemacht; dies bleibt späteren Phasen der System-Entwicklung vorbehalten.

A.b Objektorientiertes Design

Globales Design

Während in der Analyse-Phase einzelne Objekte und ihre Schnittstellen isoliert betrachtet werden, rücken im ersten Schritt der Design-Phase die Interaktionen zwischen verschiedenen Objekten in den Mittelpunkt des Interesses. Neben den Diensten, die ein Objekt anbietet, werden nun diejenigen Dienste anderer Objekte identifiziert, die es zur Aufrechterhaltung seines Zustandes und zur Bereitstellung seiner Dienste benötigt. In der Terminologie der *OOD* sind dies die *Nachrichten*, die ein Objekt an andere sendet. Diese *Client-Server*-Beziehungen zwischen Objekten lassen sich am besten in geeigneten Diagrammen ausdrücken. Eine Folge dieses Prozesses ist die Ergänzung des während der Analyse erstellten Objektwörterbuches und eine Verfeinerung der darin bereits aufgeführten Objekte:
Benötigt ein Objekt zur Erfüllung seiner Eigenschaften Dienste resp. Informationen, die bisher keinem Objekt des Objektwörterbuches zueigen sind, so kann es erforderlich sein, geeignete, vorhandene Objekte um das benötigte Verhalten zu ergänzen oder weitere Objekte zu identifizieren, die das gewünschte Verhalten aufweisen.
Solche neuen Eigenschaften vorhandener Objekte müssen ebensowenig wie die neu identifizierten Objekte in der Sicht des Anwenders vorkommen. Sie sind vielmehr Teil der *Software Requirements* des Systems und ergänzen das Objekt-Wörterbuch aus der Analyse. Dennoch können neu identifizierte Objekte auf Lücken der Analyse-Phase hindeuten. Im Zweifelsfall sind Analyse und Design mehrmals zu iterieren.

Detailiertes Design

Beim globalen Design steht die Interaktion zwischen Objekten durch Nachrichten-Übermittlung im Vordergrund. Interne Belange eines Objektes werden dazu nur soweit betrachtet, wie es zur Identifikation der von einem Objekt benötigten Dienste erforderlich ist. Im detailierten Design steht die Generalisierung von Objekten zu Klassen und der Aufbau von Klassen mit Hilfe eines Vererbungsmechanismus im Mittelpunkt. Im Einzelnen sind folgende Schritte durchzuführen:

(i) Dekomposition von Objekten in Teilobjekte
In der Analyse-Phase identifizierte Objekte der Problem-Domäne entpuppen sich bei nährer Betrachtung häufig als Strukturen, deren einzelne Komponenten eigenständige, bisher nicht identifizierte Objekte sein können. Z.B. kann eine

baum-artige Struktur als Objekt angesehen werden; ebenso können aber die Knoten und Blätter eines Baumes sinnvolle Objekte mit spezifischen Eigenschaften sein. Durch eine derartige Dekomposition von Objekten können weitere Iterationen der Analyse und des globalen Design erforderlich werden. Die Ergebnisse von Objekt-Dekompositionen bilden häufig Part-of-Hierarchien, die sich graphisch darstellen lassen.

(ii) Identifikation von Klassen

Im Allgemeinen werden einzelne Objekte des Problem-Raumes mehrfach, in Form verschiedener Instanzen auftreten. Die Beschreibungen dieser Objekte im Objekt-Wörterbuch können als vorläufige Klassendefinitionen angesehen werden, sofern die spätere Zielsprache eine strenge Trennung zwischen Klassen und Objekten bzw. die Definition einzelner Objekte überhaupt unterstützt. Da es sich hierbei mehr um begriffliche Klärung denn um eine Designmaßnahme handelt, kann dieser Schritt auch zwanglos während der Analyse vollzogen werden. Im folgenden wird angenommen, daß alle Objekte zunächst durch eine Klasse definiert werden.

(iii) Top-Down-Design von Klassen mit Hilfe von Klassen-Bibliotheken

Sind Bibliotheken mit generischen Klassendefinitionen aus frühreren Entwicklungen vorhanden, so wird man versuchen, die Eigenschaften der Klassen des Objekt-Wörterbuches mit Hilfe des Vererbungsmechanismus durch schrittweise Spezialisierung der Klassen aus Bibliotheken zu approximieren. Bei der Definition von Subklassen der generischen Klassen ist ebenfalls auf Weiterverwendbarkeit in der augenblicklichen und späteren Entwicklungen zu achten.

(iv) Bottom-Up-Design von Superklassen

Klassen des Objekt-Wörterbuches, die gar nicht oder nicht vollständig mit der Vorgehensweise aus (iii) konstruiert werden können, werden zu Gruppen ähnlicher Klassen zusammengefaßt. So können verwandte Dienste und Slots von Klassen zum Teil generalisiert und über gemeinsame Superklassen verfügbar gemacht werden.

In den Schritten (iii) und (iv) ist die graphische Darstellung der Vererbungshierarchien als hilfreiches Mittel zur Unterstüzung der Design-Phase anzusehen.
Es sei bemerkt, daß in den Unterschritten (i) bis (iv) des detailierten Design noch immer nichts über die Realisierung einzelner Methoden gesagt wird. Aufbauend auf der in (iii) und (iv) definierten Vererbungshierarchie kann nun zum Design einzelner Methoden von Klassen übergegangen werden:

(v) Programmdesign der Methoden einzelner Klassen

Da Methoden spezieller Klassen soweit möglich auf dem allgemeineren Verhalten

ererbter Methoden aufbauen sollten, muß das Design von Methoden ausgehend von allgemeinen Klassen hin zu spezielleren durchgeführt werden. Eine Methode sollte dabei nur die Funktion beinhalten, die allen gleichnamigen, spezielleren Methoden der Unterklassen ihrer Klasse gemeinsam ist. Die Unterklassen spezialisieren und erweitern diese Funktion dann für ihren speziellen Zweck.
Da Methoden die funktionalen Komponenten von Objekten sind, erfolgt ihr Design zweckmäßigerweise vermittels der *funktionalen Dekomposition*, dem globalen Entwicklungs-Paradigma der *SD*-Methoden. Dabei ergibt sich i.a. die Identifikation primitiverer Methoden eines Objektes, die nicht als Dienste in Erscheinung treten.

Der letzte Schritt des detailierten Designs realisiert einen fließenden Übergang in die Phase der objektorientierten System-Implementation. Insbesondere mündet die Entwicklung einzelner Klassen hier in einen isolierten *Object-Life-Cycle:* Design und Implementation können für Klassen bzw. Clustern von Klassen (siehe 2.2.2.3) isoliert durchgeführt werden, wobei Vererbungen in der Weise zu berücksichtigen sind, daß allgemeineren Klassen Vorrang eingeräumt wird.
Da die Schritte (iii), (iv) und (v) durchaus bereits in der Zielsprache durchgeführt werden können, mutet die Trennung der Phasen etwas künstlich an. Da sich aber dennoch unterschiedliche Schwerpunkte ergeben, wird sie hier vorgenommen.

A.c Objektorientierte Implementation

Gegenstand der Implementationsphase ist im wesentlichen die Umsetzung der in der Design-Phase spezifizierten Klassen in eine objektorientierte Zielsprache sowie das Austesten fertig programmierter Module und schließlich des gesamten Systems interagierender Klassen. Sie kann durch die nachfolgenden Schritte charakterisiert werden:

(i) Gruppierung von Klassen in Clustern
Henderson-Sellers und Edwards befürworten die von Meyer (Meyer 89) vorgeschlagene Zusammenfassung von im System stark voneinander abhängiger Klassen in Clustern, die getrennt implementiert und getestet werden können. Als Kriterium zur Bildung solcher Cluster können die in der Design-Phase ermittelten Client-Server-Beziehungen und Part-of-Hierarchien dienen. Im allgemeinen wird es nicht möglich sein, völlig unabhängige Cluster zu identifizieren; für den weiteren Verlauf der Implementation ist es dienlich, die Abhängigkeiten in einem Diagramm festzuhalten.

(ii) Implementation einzelner Klassen eines Clusters

Codierung und Test werden wie bereits angedeutet für jede Klasse weitgehend isoliert durchgeführt. Eine Klasse, deren Design vorläufig abgeschlossen ist, kann codiert und getestet werden, wenn die Implementation der Superklassen, deren Eigenschaften sie erbt, weit genug fortgeschritten ist. Klassen, die eher Dienste bereitstellen (server) haben Vorrang vor solchen, die verstärkt Dienste anderer Klassen aufrufen (clients). Entsprechend werden die Klassen einer Part-of-Hierarchie beginnend mit ihren elementarsten Bestandteilen implementiert.

(iii) Testen von Clustern

Die Codierung und das Testen einzelner Klassen eines Clusters münden in den Test des Clusters im Zusammenhang. Iterationen der Schritte (i) bis (iii) sind dabei zu erwarten.

(iv) Testen des Systems

Mit der fortschreitenden Implementierung der Cluster kann mit dem Test mehrerer, abhängiger Cluster im Verbund begonnen werden. Nach weiteren, eventuellen Wiederholungen früherer Schritte wird schließlich das gesamte System getestet.

B Dokumentation der Analyse

In den folgenden Abschnitten sind die in Kapitel 3 nur exemplarisch erläuterten Ergebnisse der Analyse-Phase weitgehend dokumentiert.

Die unten aufgeführte Sammlung der bei einer Untersuchung der integrativen Wissensakquisitionsmethode identifizierbaren Objekte erhebt keinen Anspruch auf Vollständigkeit. Die Identifikation weiterer Objekte während der Design- und Implementations-Phase ist möglich. Allerdings basiert die Erstellung des Objektwörterbuches auch auf Erfahrungen aus einer prototypischen Implementation der integrativen Wissensakquisitionsmethode. Es sind daher eine Reihe von Aspekten enthalten, die im Normalfall einer objektorientierten Systementwicklung erst in den einzelnen Schritten der Design-Phase aufgedeckt werden. Ein Beispiel hierfür ist die Dekomposition der Klasse *Explanation* in die Klassen *Explanation* und *Sub-Explanation:* Instanzen der Klasse *Explanation* dienen als Wurzel einer Erklärungshierarchie und verwalten eine Erklärung zu einem Fall. Die Instanzen der Klasse *Sub-Explanation* hingegen verweisen auf einzelne Knowledge-Units, die den erklärenden Text beinhalten.
Mit (AK) gekennzeichnete Klassen, Slots oder Dienste werden nur aus Gründen der AufwärtsKompatibilität für den Fall späterer Verfeinerungen der integrativen KA-Methode und entsprechenden Erweiterungen der Implementation erwähnt. Sie brauchen in Design und Implementation nicht unbedingt berücksichtigt werden.

Klasse *Informal-Knowledge-Base* - Slots

Slots	Beschreibung
Case-References	Da die Fälle zwecks Verwendung in verschiedenen informalen Wissensbasen nicht deren Bestandteil sind, werden Referenzen auf sie verwaltet.
Creator (AK)	Der Anwender, der eine informale Wissensbasis angelegt hat. Er kann aus Gründen der Aufwärtskompatibilität hinsichtlich eines Mehrbenutzer-Zugriffes auf eine informale Wissensbasis mitverwaltet werden.
Domain-Model	Eine informale Wissensbasis ist mit einem Modell der Domäne assoziiert. Die Persistenz dieses Modelles ist für die Integrität der informalen Wissensbasis mit ausschlaggebend. Es soll daher von dieser verwaltet werden.
Expertise-Model	Das für das Domänen-Modell Gesagte gilt in gleicher Weise für für das Modell der Expertise.
Explanations	In COKAM+ werden zu verschiedenen Fällen Erklärungsstrukturen erhoben, die Bestandteil einer informalen Wissensbasis sind.
Features	Einen weiteren Bestandteil bilden Features, die durch CECoS erhoben werden.
Title	Der Name einer informalen Wissensbasis.
Production-Class-Hierarchy	Ebenfalls als Output von CECoS enthält eine informale Wissensbasis eine Hierarchie von Produktionsklassen.
Source-Text-References	Quelltexte sind nicht Bestandteil der informalen Wissensbasis. Es werden aber Referenzen auf Quelltexte verwaltet, die es erlauben sollen, die Textquelle einer Unit jederzeit identifizieren zu können.
Unit-Selectors	Bei der Selektion von Units und Features einer informalen Wissensbasis können eine Reihe von Selektionskriterien spezifiziert werden. Um die wiederholte Angabe gleicher Kriterien zu vermeiden und eine inhaltliche Gruppierung von Units zu erlauben, sollen einmal angegebene Selektionskriterien in Macros zusammengefaßt werden können. Der Slot *Unit-Selectors* hat als Wert eine Liste solcher Macros.
Units	Sämtliche *Knowledge-Units* einer Wissensbasis.

Klasse *Informal-Knowledge-Base* - Dienste

Dienste	
create-unit-selector	Erzeugt ein Macro zur Selektion von Units oder Features aus anzugebenden Selektionskriterien (siehe Klasse *Unit-Selector*).
generate-case-pairlist	Dieser Dienst liefert eine Liste sämtlicher 2-Kombinationen der durch *Case-References* referenzierten Fälle. Jede dieser Kombinationen ist ein nicht persistentes Objekt vom Typ *Case-Pair*.
new-case-reference	Erzeugt ein neues Objekt vom Typ *Case-Reference* und assoziiert es mit einem angegebenen Fall.
new-explanation	Erzeugt ein neues Objekt vom Typ *Explanantion* und assoziiert es mit einer *Case-Reference*
new-feature	Liefert ein neues Objekt vom Typ *Feature*
new-unit	Stellt ein neues Objekt vom Typ *Knowledge-Unit* bereit.
select-units	Liefert sämtliche Knowledge-Units oder Features, die anzugebenden Kriterien genügen.
show-domain-model, *show-expertise-model*	Bringt das zu einer informalen Wissensbasis gehörende Domänen- bzw. Expertisen-Modell zur Anzeige.
set-title	Setzen/Ändern des Namens

Klasse *Case*

Slots	Beschreibung
Mold (AK)	Beschreibung eines Rohteils
Production-Plan (AK)	Ein Produktions-Plan
Title	Titel des Falles.
Workpiece-Model	Produkt-Modell, zuächst nur eine Graphik.
Workshop (AK)	Kann ein Objekt vom Typ Workshop enthalten, das eine Arbeitsumgebung (Maschinen, Werkzeuge etc.) modelliert.
Dienste	
set-title	setzt den Titel eines Falles
delete	löscht den Fall
save	speichert eine Fall-instanz
show	Anzeigen der Graphik des Slots Workpiece-Model.

Klasse *Case-Base*

Eine Instanz der Klasse Case-Base organisiert sämtliche der Wissensakquisition zur Verfügung stehenden Fälle.

Slots	Beschreibung
Cases	Eine Liste mit Instanzen der Klasse Case
Dienste	
select-case	Erlaubt einem Anwender, Fälle aus der Fallbasis für eine informale Wissensbasis auszuwählen.

Klasse *Case-Pair*

Zur Erzeugung der Hierarchie der Produktionsklassen werden in CECoS Paare von Fällen gebildet, die mit einem vom Experten anzugebenden Wert für die Ähnlichkeit zweier in einem Paar zusammengefaßter Fälle assoziiert werden. Die Paare werden von einer informalen Wissensbasis im Rahmen des Dienstes *generate-pairlist* (siehe Klasse *Informal-Knowledge-Base*) erzeugt und haben nur temporäre Bedeutung.

Slots	Beschreibung
Case_1, Case_2	Zwei Objekte vom Typ *Case*. Ein Case-Pair ist eine 2-Kombination der durch den Slot *Case-References* einer informalen Wissensbasis referenzierten Fälle.
Similarity	Die durch einen Experten zu taxierende Ähnlichkeit der Fälle aus den Slots *Case_1* und *Case_2* hinsichtlich ihres Produktionsplanes.
Dienste	
get-similarity	Ein als natürliche Zahl übergebens Argument wird Wert des Slots *Similarity*.
hide	Zwei gleichzeitig, graphisch dargestellte Fälle, die dem Paar angehören, verschwinden vom Bildschirm.
show	Die mit dem Paar assoziierten Fälle werden gleichzeitig, an benachbarter Position graphisch dargestellt.

Klasse *Case-Pair-List*

Ein Objekt dieser Klasse wird durch den Dienst generate-pairlist einer informalen Wissensbasis erzeugt und enthält alle 2-Kombinationen der durch *Case-References* referenzierten Fälle.

Slots	**Beschreibung**
Pair-List	Liste von Objekten des Types *Case-Pair*.
Dienste	
current-pair	Das zur Zeit durch den Benutzer bearbeitete Element der Pair-List.
serve-pair-comparison	Ein den Paarvergleich vorbereitender Dienst, etwa die Aufforderung der graphischen Benutzerschnittstelle zur Darstellung des entsprechenden Dialoges.
show-next-pair	Bringt das auf den Wert des Slots *Current-Pair* folgende Paar zur Darstellung und macht dieses zum *Current-Pair*. Hat der Slot Current-Pair zur Zeit des Aufrufes von *show-next-pair* keinen Wert, wird als Default das erste Paar der Liste angezeigt. Die Anzeige erfolgt durch Aufruf des Dienstes *show* eines Objektes *Case-Pair*.
show-previous-pair	Die zu show-next-pair inverse Aktion.
terminate-pair-comparison	Beendet den Paarvergleich und entfernt die assoziierten Bildschirmobjekte.
generate-production-class-hierarchy	Nach vollzogenem Paarvergleich wird mittels einer hierarchischen Clusteranalyse (Tschaitschian 91) ein Objekt der Klasse *Production-Class-Hierarchy* (siehe dort) sowie damit assoziierte Objekte erzeugt.

Klasse *Case-Reference*

Da verschiedene Wissensbasen auf gleiche Fälle Bezug nehmen können, werden dort nur Referenzen auf Objekte des Typs *Case* verwaltet.

Slots	Beschreibung
Case	Kann ein durch *Case-Location* referenziertes Objekt enthalten.
Location	Eine Referenz, die besagt, von wo ein Fall zu laden ist, etwa ein Pfadname.
Dienste	
show	Läd den durch *Location* referenzierten Fall nach *Case*
load-case	Zeigt den in *Case* enthaltenen Fall, oder eine Meldung, falls der Slot *Case* keinen Wert hat.

Klasse *Domain-Model-Category*

Die Objekte dieser Klasse sind Bestandteile von Domänen-Modellen (siehe folgende Klasse). Zu ihr können Subklassen definiert werden, die die einzelnen Kategorien einer Anwendungsdomäne widerspiegeln.

Slots	Beschreibung
Sub-Categories	Die Nachfolger in der Hierarchie des Domänen-Modelles.
Super-Category	Vorgänger im Domänen-Modell. Dies ist entweder ein Objekt der Klasse *Domain-Model-Category* oder ein *Domain-Model*
Title	Der Name einer Domänen-Kategorie.
Dienste	
create-sub-category	Erzeugt einen Nachfoger vom Typ *Domain-Model-Category*.
highlight	Die angesprochene Kategorie soll in der Darstellung des Domänen-Modelles hervorgehoben werden
set-title	Setzen/Ändern des Titels

Klasse *Domain-Model*

Ein Domänen-Modell soll als Baumstruktur dargestellt werden. Ein Objekt der Klasse *Domain-Model* kann Wurzel eines solchen Baumes sein. Die übrigen Knoten sind Objekte der Klasse *Domain-Model-Category*.

Slots	**Beschreibung**
Sub-Categories	Die direkten Nachfolger der Wurzel des Domänen-Modell-Baumes.
Title	Der Name des Domänen-Modelles.
Dienste	
create-sub--category	Erzeugt einen Nachfoger vom Typ Domain-Model-Category.
hide	Entfernen des Domänen-Modelles vom Bildschirm
save	Speichern eines Domänen-Modelles ausserhalb der informalen Wissensbasis zum Zwecke der Weiterverwendung in anderen informalen Wissensbasen
set-title	wie üblich.
show	Darstellen des Domänen-Modelles

Klasse *Expertise-Model-Category*

Diese Klasse verhält sich analog zur Klasse *Domain-Model-Category*, siehe dort.

Klasse *Expertise-Model*

Die Struktur dieser Klasse gleicht der der Klasse *Domain-Model* ; inhaltliche Unterschiede werden erst durch den Anwender spezifiziert.

Klasse *Explanation*

Die Erklärung eines Falles durch einen Experten wird in COKAM+ durch eine Baumstruktur repräsentiert. Ein Objekt vom Typ *Explanation* ist die Wurzel einer Erklärungstruktur. Die Nachfolger sind vom Typ *Explanation-Node*.

Slot	Beschreibung
Case-Reference	Ein Verweis auf den durch die mit einem Objekt *Explanation* verbundene Struktur erklärten Fall, ein Objekt der Klasse *Case-Reference*.
Date	Erzeugungsdatum der Erklärung.
Expert (AK)	Der Urheber der Erklärung - gewöhnlich ein Experte.
Sub-Explanations	Die direkten Nachfolger der Wurzel der Erklärungsstruktur.
Title	
Dienste	
create-sub-explanation	Ein Objekt vom Typ Sub-Explanation wird erzeugt und mit der aktuellen Unit assoziiert. Diese liefert der Dienst *current-unit* des *Unit-Stack*. Das erzeugte Objekt wird dem Inahlt des Attributes *Sub-Explanations* hinzugefügt.
delete	Löschen einer Erklärungsstruktur
hide	entfernt eine in einem Fenster dargestellte Erklärungsstruktur vom Schirm
set-title	Setzen eines Titels
show	Die Erklärungsstruktur wird auf dem Schrim angezeigt.

Klasse *Feature*

Slots	Beschreibung
Domain Category	Eine dem Feature zugewiesene Kategorie des Domänen-Modells.
Expertise Category	Eine Kategorie des Modells der Expertise, die dem Feature zugewiesen wurde.
Text	Die Beschreibung eines Features
Title	Der Titel eines Features.
Dienste	Diese Dienste haben die übliche, ihrem Namen entsprechende Bedeutung:
set-title	
delete	
hide	
show	

Klasse *Production-Class*

Slots	Beschreibung
Sub-Classes-or-Cases	Eine Productionclass kann Oberklasse mehrerer, speziellerer Produktionsklassen sein und/oder prototypische Fälle enthalten
Super-Class	Eine abstraktere Produktionsklassse
Title	wie üblich
Dienste	
check-consistence	Überprüft die in CECoS gültigen Konsistenzregeln für Produktionsklassen-Hierarchien.
delete	Löschen einer Produktionsklasse aus der Hierarchie
paste	Einfügen einer zuvor gelöschten Produktionsklasse in die Hierarchie

Klasse *Production-Class-Hierarchy*

Slots	Beschreibung
Creator	Der Erzeuger einer Production-Class-Hierarchy
Dienste	wie üblich:
delete	
hide	
show	

Klasse *Production-Plan (AK)*

Spätere Versionen des Wissensakquisitionssystems sollten in der Lage sein, Arbeitspläne adequat zu repräsentieren.

Klasse *Source-Text*

Beschreibung siehe Kapitel 3.2

Klasse *Source-Text-Reference*

Slots	Beschreibung
Location	Ein Pfadname, eventuell mit Angabe eines des Rechnerknotens
Source-Text	Der Titel des Textes
Dienste	
load-sourcetext	Läd einen Quelltext aus *Location*
show-reference	Zeigt den Namen des Quelltextes.

Klasse *Sub-Explanation*

Die für einen Fall gegebene Erklärung wird durch ein Objekt der Klasse *Explanation* repräsentiert, das als Wurzel des Erklärungsbaumes dient. Die weiteren Knoten des Baumes werden durch Objekte der Klasse *Sub-Explanation* repräsentiert

Slots	Beschreibung
Knowledge-Unit	Ein Verweis auf eine Wissenseinheit
Sub-Explanations	Weitere Instanzen der Klasse Sub-Explanation
Dienste	
create-sub-explanation	Siehe Klasse Explanation
delete	Löschen der Sub-Explanation aus dem Erkklärungsbaum
validate-sub-explanation	Prüfen der Konsistenz des Erklärungsbaumes für eine neu hinzugekommene Instanz der Klasse Sub-Explanation

Klasse *Unit*

Beschreibung siehe Kapitel 3.2

Klasse *Unit-Stack*

Slots	Beschreibung
Visible-Units	Liste der zur Zeit im Kartenstapel befindlichen Units (diese müssen nicht unbedingt sichtbar sein)
Dienste	
current-unit	Die zur Zeit in Bearbeitung befindliche Unit
new-current-unit	Eine als Argument übergebene Unit wird zur *current-unit*
sort	Der Kartenstapel wird nach anzugebenden Kriterien sortiert

Klasse *Workshop (AK)*

Slots	Beschreibung
Machines	Verfügbare Drehmaschinen
Tools	Verfügbare Werkzeuge
Dienste	

Klasse *Workpiece-Model*

Diese Klasse modelliert das in eine Problemlösung eingehenden Arbeitsumfeld:

Slot	Beschreibung
Geometrical-Data (AK)	Geometrische Werkstückdaten
Picture	Graphische Darstellung
Technological-Data (AK)	Technologische Werkstückdaten
Dienste	wie üblich:
show	
hide	
delete	

C Benchmarks

Um einen ungefähren Eindrucd von der Leistungsfähigkeit der in Kapitel 5.1 besprochenen Maschinen in Verbindung mit den entsprechenden Common Lisp/CLOS Umgebungen zu erhalten, wurde ein einfacher Benchmark-Test durchgeführt, der die Effizienz eines Aufrufs einer generischen Funktion mit einer Reihe von Methoden ermittelt. Die in der Tabelle C.1 angegebenen Zeiten sind Mittelwerte.

Common Lisp/CLOS - Benchmark Test

Maschine	Lisp/CLOS	Zeit
SPARCstation SLC	AKCL/PCL	0.230 sec
SPARCserver 4/390	AKCL/PCL	0.205 sec
Symbolics Ivory Board	GENERA 8.0.2/CLOS	**0.067 sec**
Macintosh II c	Mac Allegro CL/PCL	0.687 sec
Macintosh II fx	Mac Allegro CL/PCL	**0.149 sec**

Tabelle C.1: Benchmarks für Lisp/CLOS-Umgebungen auf den bei ARC-TEC im Einsatz befindlichen Maschinen.

D Code-Beispiele

Die folgenden CLOS-Methoden `prepare-text`, `status-function` und `recognize` der Klasse `lsi-process` gebrauchen die ererbte Methode `rcall`. Man beachte, wie das durch einen Non-blockingAufruf von `rcall` erzeugte Closure (`prep-text-closure`) mittels einer vordefinierten status-function in ein weiteres Closure eingebettet wird. Der Slot `status-function` erhält dieses letztere Closure als Wert. Es informiert den Benutzer zu gegebener Zeit über die fertige Preparation eines Quelltextes, indem z.B. die Lisp-Top-Level-Loop es wiederholt durch `(funcall (status-function ls-process-instance))` aufruft.

```
(defmethod prepare-text ((instance lsi-process) (filename `pathname))
  (unless (recognize instance filename)
    (let ((prep-text-closure (rcall `prepare-text instance :nonblocking t)))
      (with-open-file (file filename :direction :input)
        (copy-character-stream file
                               (ostream instance)
                               :eof-value "#E"))
      (setf (lsi-status instance) :prepare)
      (setf (status-function instance)
            (function (lambda ()
                        (status-function instance (lsi-status instance)
                                         prep-text-closure filename)))))))


(defmethod status-function ((instance lsi-process) (status (eql :prepare))
                            (closure `function) (filename `pathname))
  (when (data-arrived-p instance)
    (let ((result (funcall prep-text-closure)))
      (cond ((eq result :ok)
             (screen-message
              (format nil "~A is ready for LSI-Search now!"
                      file-name)))
            ((eq result :error)
             (screen-message
              (format
               nil
               "An error occurred during the preparation of ~A"
               filename)))
            (t
             (screen-message
              "Fatal error: Unknown result code"))))))


(defmethod recognize ((instance lsi-process) (filename `string))
  (rcall (concatanate `string "recognize "
                              filename)
         instance))
```

Literatur

(Birk 91)
Birk, A.: *Verifikation von Wissen während der Akquisitionsphase am Beispiel COKAM.* Projektarbeit am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI), Kaiserslautern 1991.

(Bobrow 88)
Bobrow, Daniel G. et al.: *The Common Lisp Object System Specification,*(X3J13-88-002). American National Standards Institut, 1988.

(Booch 86)
Booch, G.: *Object Oriented Development.* IEEE Transactions on Software Engineering, Feb. 1986.

(CLUE)
Common Lisp User Interface Environment. N.N.

(Cutts 88)
Cutts, G.: *Structured systems analysis and design methodology.* Van Nostrand Reinhold Company, New York 1988.

(Dumais 88)
Dumais, S.T.; Furnas, G.W.; Landauer, T.K; Deerwester, S; Harshman, R.: *Using Latent Semantic Analysis to improve Access to textual information.* CHI 88 Proceedings, 1988.

(Edwards, King, Winblad 90)
Edwards, S.; Winblad, A.; King, D.:*Objekt-orinentierte System Entwicklung.*Reading, Mass. Edison Wesley, 1990.

(Henderson-Sellers, Edwards 90)
Henderson-Sellers, Brian; Edwards, Julian M.: *The Object-Oriented Systems Life Cycle.* Communications of the ACM, Sep. 1990.

(Jackson 83)
Jackson, M.A.:, *System Development,* Prentice Hall, London 1983.

(Keene 89)
Keene, Sonya E.: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Reading, Mass.: Addison-Wessley, 1989.

(Kieras 91)
Kieras, D.: *A computer-based aid for compehensible writing.* Präsentiert auf der 2. jährlichen Winter text conference, 1991.

(Laufkötter 91)
Laufkötter, G.: *COKAM+ User Manual,* ARC-TEC-Diskussionspapier Nr. 91-12, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Kaiserslautern 1991.

(Loy 90)
Loy, P.: *A Comparison of Object-Oriented And Structured Development Methods,* ACM SIGSOFT Software Engineering Notes vol. 15, Jan 1990.

(Parnas 72)
Parnas, D.: *On the criteria to be used in decomposing systems into modules.* Communications of the ACM, Dec. 1972.

(Pfaff 85)
Pfaff, G.E.: *User Interface Management Systems.* Springer Verlag, Berlin, 1985.

(Schmalhofer, Kühn et al.90)
Schmalhofer, F;Kühn,O.;Schmidt, G.: Integrated *knowledge acquisition from texts, previously solved cases and expert memories.* In Applied Artificial Intelligence Volume 5, 1991.

(Schmalhofer, Schmidt 91)
Schmalhofer, F; Schmidt, G, *Situated Text-Analysis with COKAM+.* In European Knowledge Acquisition Workshop, Sisyphus Working Papers: Text Analysis. J. Boose, B. Gaines, M. Linster, D. Smeed and B. Woodward, 1991.

(Schmidt 91)
Schmidt, G.: *COKAM+: Fall-orientietre und Modell-orientierte Wissensakquisition aus Texten.* ARC-TEC-Diskussionspapier Nr 91-07,Deutsches Forschungszentrum für künstliche Intelligenz (DFKI) Kaiserslautern, 1991.

(Sechrest 87)
Sechrest, S.: *An Introductory 4.3 BSD Interprocess Communication Tutorial*, Berkeley, 1988.

(Shlaer, Mellors 89)
Shlaer,S; Mellor, S.: *Object-Oriented Systems Analysis: Modelling the World in Data.* Yourdon Press Computing Series, 1988.

(Sloman 88)
Sloman, Morris: *Verteilte Systeme und Rechnernetze*, Hansa Verlag Münnchen, Prentice Hall London, 1988.

(Steele 84)
Steele, Guy: *Common Lisp: The Language.* Maynard, Mass.: Digital Press, 1984.

(Steele 89)
Steele, Guy: *Common Lisp: The Language.* Maynard, Mass.: Digital Press, 1989.

(Tschaitschian 90)
Tschaitschian, B: *Eine integrative Wissenserhebung und -analyse mit CECoS: Konzepte und prototypische Implementierung*, Projektarbeit am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI), Kaiserslautern 1990.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all publisched papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist: WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta: RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke: A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel: Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-12
J.Mark Gawron, John Nerbonne, Stanley Peters: The Absorption Principle and E-Type Anaphora
33 pages

RR-91-13
Gert Smolka: Residuation and Guarded Rules for Constraint Logic Programming
17 pages

RR-91-14
Peter Breuer, Jürgen Müller: A Two Level Representation for Spatial Relations, Part I
27 pages

RR-91-15
Bernhard Nebel, Gert Smolka: Attributive Description Formalisms ... and the Rest of the World
20 pages

RR-91-16
Stephan Busemann: Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

RR-91-17
Andreas Dengel, Nelson M. Mattos: The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

RR-91-18
John Nerbonne, Klaus Netter, Abdel Kader Diagne, Ludwig Dickmann, Judith Klein: A Diagnostic Tool for German Syntax
20 pages

RR-91-19
Munindar P. Singh: On the Commitments and Precommitments of Limited Agents
15 pages

RR-91-20
Christoph Klauck, Ansgar Bernardi, Ralf Legleitner FEAT-Rep: Representing Features in CAD/CAM
48 pages

RR-91-21
Klaus Netter: Clause Union and Verb Raising Phenomena in German
38 pages

RR-91-22
Andreas Dengel: Self-Adapting Structuring and Representation of Space
27 pages

RR-91-23
Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

RR-91-24
Jochen Heinsohn: A Hybrid Approach for Modeling Uncertainty in Terminological Logics
22 pages

RR-91-25
Karin Harbusch, Wolfgang Finkler, Anne Schauder: Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

RR-91-26
M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger: Integrated Plan Generation and Recognition - A Logic-Based Approach -
17 pages

RR-91-27
A. Bernardi, H. Boley, Ph. Hanschke, K. Hinkelmann, Ch. Klauck, O. Kühn, R. Legleitner, M. Meyer, M. M. Richter, F. Schmalhofer, G. Schmidt, W. Sommer: ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge
18 pages

RR-91-28
Rolf Backofen, Harald Trost, Hans Uszkoreit: Linking Typed Feature Formalisms and Terminological Knowledge Representation Languages in Natural Language Front-Ends
11 pages

RR-91-29
Hans Uszkoreit: Strategies for Adding Control Information to Declarative Grammars
17 pages

RR-91-30
Dan Flickinger, John Nerbonne: Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39 pages

RR-91-31
H.-U. Krieger, J. Nerbonne: Feature-Based Inheritance Networks for Computational Lexicons
11 pages

RR-91-32
Rolf Backofen, Lutz Euler, Günther Görz: Towards the Integration of Functions, Relations and Types in an AI Programming Language
14 pages

RR-91-33
Franz Baader, Klaus Schulz: Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures
33 pages

RR-91-34
Bernhard Nebel, Christer Bäckström: On the Computational Complexity of Temporal Projection and some related Problems
35 pages

RR-91-35
Winfried Graf, Wolfgang Maaß: Constraint-basierte Verarbeitung graphischen Wissens
14 Seiten

RR-92-01
Werner Nutt: Unification in Monoidal Theories is Solving Linear Equations over Semirings
57 pages

RR-92-02
Andreas Dengel, Rainer Bleisinger, Rainer Hoch, Frank Hönes, Frank Fein, Michael Malburg: Π_{ODA}: The Paper Interface to ODA
53 pages

RR-92-03
Harold Boley: Extended Logic-plus-Functional Programming
28 pages

RR-92-04
John Nerbonne: Feature-Based Lexicons: An Example and a Comparison to DATR
15 pages

RR-92-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Michael Schulte, Rainer Stark: Feature based Integration of CAD and CAPP
19 pages

RR-92-07
Michael Beetz: Decision-theoretic Transformational Planning
22 pages

RR-92-08
Gabriele Merziger: Approaches to Abductive Reasoning - An Overview -
46 pages

RR-92-09
Winfried Graf, Markus A. Thies: Perspektiven zur Kombination von automatischem Animationsdesign und planbasierter Hilfe
15 Seiten

RR-92-11
Susane Biundo, Dietmar Dengler, Jana Koehler:
Deductive Planning and Plan Reuse in a Command Language Environment
13 pages

RR-92-13
Markus A. Thies, Frank Berger:
Planbasierte graphische Hilfe in objektorientierten Benutzungsoberflächen
13 Seiten

RR-92-14
Intelligent User Support in Graphical User Interfaces:
1. InCome: A System to Navigate through Interactions and Plans
   Thomas Fehrle, Markus A. Thies
2. Plan-Based Graphical Help in Object-Oriented User Interfaces
   Markus A. Thies, Frank Berger

22 pages

RR-92-15
Winfried Graf: Constraint-Based Graphical Layout of Multimodal Presentations
23 pages

RR-92-17
Hassan Aït-Kaci, Andreas Podelski, Gert Smolka:
A Feature-based Constraint System for Logic Programming with Entailment
23 pages

RR-92-18
John Nerbonne: Constraint-Based Semantics
21 pages

DFKI Technical Memos

TM-91-01
Jana Köhler: Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann: Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt:
Clamping, COKAM, KADS, and OMOS:
The Construction and Operationalization of a KADS Conceptual Model
20 pages

TM-91-04
Harold Boley (Ed.):
A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel , Rainer Bleisinger:
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

TM-91-06
Johannes Stein: Aspects of Cooperating Agents
22 pages

TM-91-08
Munindar P. Singh: Social and Psychological Commitments in Multiagent Systems
11 pages

TM-91-09
Munindar P. Singh: On the Semantics of Protocols Among Distributed Intelligent Agents
18 pages

TM-91-10
Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch: Tree Adjoining Grammars mit Unifikation
149 pages

TM-91-11
Peter Wazinski: Generating Spatial Descriptions for Cross-modal References
21 pages

TM-91-12
Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13
Knut Hinkelmann:
Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Bussmann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang:
Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

DFKI Documents

D-91-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

D-91-06
Gerd Kamp: Entwurf, vergleichende Beschreibung und Integration eines Arbeitsplanerstellungssystems für Drehteile
130 Seiten

D-91-07
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

D-91-08
Thomas Krause: Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 Seiten

D-91-09
David Powers, Lary Reeker (Eds.):
Proceedings MLNLO'91 - Machine Learning of Natural Language and Ontology
211 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-10
Donald R. Steiner, Jürgen Müller (Eds.):
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-11
*Thilo C. Horstmann:*Distributed Truth Maintenance
61 pages

D-91-12
Bernd Bachmann:
$H^{iera}C_{on}$ - a Knowledge Representation System with Typed Hierarchies and Constraints
75 pages

D-91-13
International Workshop on Terminological Logics
Organizers: Bernhard Nebel, Christof Peltason, Kai von Luck
131 pages

D-91-14
Erich Achilles, Bernhard Hollunder, Armin Laux, Jörg-Peter Mohren: KRIS: Knowledge Representation and Inference System
- Benutzerhandbuch -
28 Seiten

D-91-15
Harold Boley, Philipp Hanschke, Martin Harm, Knut Hinkelmann, Thomas Labisch, Manfred Meyer, Jörg Müller, Thomas Oltzen, Michael Sintek, Werner Stein, Frank Steinle:
µCAD2NC: A Declarative Lathe-Worplanning Model Transforming CAD-like Geometries into Abstract NC Programs
100 pages

D-91-16
Jörg Thoben, Franz Schmalhofer, Thomas Reinartz:
Wiederholungs-, Varianten- und Neuplanung bei der Fertigung rotationssymmetrischer Drehteile
134 Seiten

D-91-17
Andreas Becker:
Analyse der Planungsverfahren der KI im Hinblick auf ihre Eignung für die Abeitsplanung
86 Seiten

D-91-18
Thomas Reinartz: Definition von Problemklassen im Maschinenbau als eine Begriffsbildungsaufgabe
107 Seiten

D-91-19
Peter Wazinski: Objektlokalisation in graphischen Darstellungen
110 Seiten

D-92-01
Stefan Bussmann: Simulation Environment for Multi-Agent Worlds - Benutzeranleitung
50 Seiten

D-92-08
Jochen Heinsohn, Bernhard Hollunder (Eds.):
DFKI Workshop on Taxonomic Reasoning Proceedings
56 pages

D-92-09
Gernod P. Laufkötter: Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes
86 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes

D-92-09
Document

Gernod P. Laufkötter